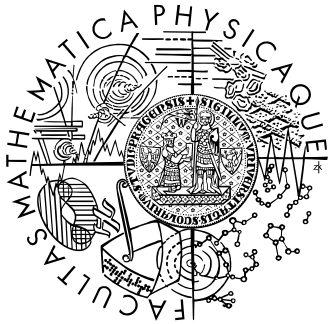
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Miroslav Cicko

Profiler jazyka C#

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

Studijní program: Informatika, programování

2008

Ďakujem pánovi RNDr. Filipovi Zavoralovi, Ph.D. za odborné vedenie tejto práce, za rady a za čas, ktorý mi počas vypracovávania tejto práce venoval.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce.

V Prahe dňa Miroslav Cicko

Obsah

Název práce: Profiler jazyka C#

Autor: Miroslav Cicko

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

e-mail vedoucího: Filip.Zavoral@mff.cuni.cz

Abstrakt:

Programovací jazyk C# poskytuje silné jazykové prostriedky a pritom je veľmi jednoduché sa ho naučiť. Spolu s platformou .NET Framework umožňuje rýchlo a pohodlne vyvíjať programy. Avšak jeho použitie ešte nezaručuje vytvorenie efektívneho programu. Cieľom tejto práce je vytvorenie profilera merajúceho trvanie a spotrebovanú pamäť vybraných príkazov. Aplikácia profilera je vybudovaná s využitím knižnice Code Toolkit vznikajúcej súčasne s profilerom. Táto knižnica poskytuje parsovanie zdrojového kódu do objektového modelu, jeho modifikáciu a následné uloženie naspäť do zdrojového súboru. Knižnica môže byť znovu použitá pri tvorbe ďalších nástrojov pracujúcich so zdrojovým programom v jazyku C#.

Klíčová slova: profiler, C#, výkonnosť

Title: C# Profiler

Author: Miroslav Cicko

Department: Department of software engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Supervisor's e-mail: Filip.Zavoral@mff.cuni.cz

Abstract:

The C# programming language provides powerful language features and it is very easy to get to learn, as well. Together with .NET Framework it allows programmers to develop applications quickly and comfortably. However, using this language does not still guarantee creation of an effective programme. The purpose of this work is to develop a profiler measuring elapsed time and consumed memory of the chosen statements. The profiler application is built with heavy usage of the Code Toolkit library which was created as part of this work, as well. This library provides facilities for parsing C# source code into an object model representation, modification and saving it back to the source files. The library can be re-used in the development of the various tools processing source code of the C# language.

Keywords: profiler, C#, performance

1. Úvod

Moderný programovací jazyk C# sa teší stále väčšej popularite vďaka svojej jednoduchej a prehľadnej syntaxe, ktorá spojuje množstvo užitočných techník rôznych programovacích jazykov ako C++, Java, Delphi a hlavne v poslednej dobe i jazykov funkcionálneho programovania. Zrozumiteľná a stručná syntax s podporou komponentového programovania s automatickou správou pamäte a podporou deklaratívneho programovania pomocou atribútov sú všetko výhody tohto jazyka. Jeho prednosti sú navyše posilnené bohatou knižnicou platformy .NET Framework, na ktorej je jazyk C# postavený. Vysoko produktívny jazyk a spätosť s behovým prostredí CLR (Common Language Runtime .NET Frameworku) s automatickou správou pamäte[2] nemusí znamenať len samé výhody. Naopak, nesprávne zaobchádzanie môže viesť k neefektívnemu behu programu a zbytočnému plytvaniu pamäte. Zvolená implementácia môže tiež trpieť výkonnostnými nedostatkami. V takomto prípade môže použitie vhodného nástroja výrazne pomôcť k nájdeniu kritických častí programu a práve vytvorenie takéhoto nástroja je jedným z cieľov tejto bakalárskej práce.

1.1 Vývoj zamerania tejto práce

Už existuje niekoľko nástrojov na hľadanie problémov s efektivitou, avšak profiler vytvorený v tejto práci používa riešenie, v ktorom je vlastné meranie uskutočnené vkladaním kódu do pôvodného programu v jazyku C#. Odôvodnením a analýzou tohto prístupu sa zaoberá 2. kapitola, avšak na realizáciu takéhoto riešenia bolo potrebné vytvoriť podpornú knižnicu umožňujúcu manipuláciu so zdrojovým kódom. Podporná knižnica Code Toolkit sa začala ukazovať ako veľmi silný a pohodlný nástroj nie len pre účely profilera, ale i v úlohe univerzálnej knižnice poskytujúcej významné zjednodušenie tvorby ľubovoľného nástroja pracujúceho so C# kódom na statickej, či behovej úrovni. Vzhľadom na objavený potenciál bola väčšia časť pozornosti počas vývoja venovaná práve tejto knižnici a samotná aplikácia

profilera sa tak stala len demonštratívnym programom poukazujúcim na výhodnosť jej použitia.

1.2 Prehľad kapitol

Pre jednoduchšiu orientáciu je zostavený nasledujúci stručný prehľad kapitol poskytujúci jednak rýchly prehľad o tom, čo všetko zahŕňa text práce, ale i prípadnú pomoc pri hľadaní žiadanej témy.

V nasledujúcej **2. kapitole** sú uvedené rôzne možnosti realizácie profilera. Táto kapitola tiež odôvodňuje, prečo bol v tejto práci zvolený práve vybraný spôsob profilovania a porovnáva jeho výhody a nevýhody s inými spôsobmi.

Prehľadu iných existujúcich profilerov sa venuje **3. kapitola**, ktorá poukazuje na ich zaujímavé vlastnosti, aby si čitateľ mohol vytvoriť širší obraz o tom, čo všetko môže pojem profilovanie zahŕňať.

Používanie profilera, ktorý je výsledkom tejto práce, je potom popísané v **4.kapitole**. Táto kapitola nepredstavuje len popis používania, ale i na príkladoch ukazuje rôzne typy výkonnostných, či pamäťovým problémov a poskytuje návod ako správne použiť profiler na ich odhalenie.

Vnútorné fungovanie knižnice Code Toolkit je vysvetlené v rozsiahlej **5. kapitole**, ktorá najprv predstavuje celkový koncept knižnice Code Toolkit a potom postupne vysvetľuje jej jednotlivé komponenty.

Implementácia samotnej aplikácie profilera je predstavená v **6.kaptole**. Aplikácia v plnom rozsahu využíva knižnicu CodeToolkit, preto popis v tejto kapitole nadväzuje na funkcionalitu opísanú v 5. kapitole.

7. kapitola predstavuje záver zhrňujúci vytvorenú prácu a naznačuje možnosti jej rozšírenia v budúcnosti.

2 Analýza problému profilovania

Profilovať je možné rôzne veličiny a to rôznymi spôsobmi. Je preto potrebné si stanoviť, čo všetko bude objektom merania. Ale najprv začnem analýzou možných spôsobov ako vôbec meranie realizovať.

2.1 Spôsoby profilovania

Základná myšlienka spôsobu merania je veľmi jednoduchá. Spotrebovaný čas resp. pamäť sa zmeria pred príkazom a po ňom a z rozdielu týchto dvoch hodnôt sa určí spotreba daného príkazu. Zdanlivo jednoduché riešenie však v sebe skrýva náročný problém vkladania príkazov (inštrumentácia[4]) na správne miesto v zdrojovom kóde, ale i ďalšie problémy súvisiace s komunikáciou rozhrania profilera s profilovanou aplikáciou a synchronizáciou ukladania a čítania nameraným údajov. Existujú aj iné spôsoby realizácie profilera – menovite využitie .NET Profiling API[4], či inštrumentácia CIL (tiež známeho ako MSIL[2]) kódu vytvoreného kompiláciou .NET jazyka. Tieto spôsoby majú svoje výhody i nevýhody oproti inštrumentácii zdrojového kódu, avšak metóda inštrumentácie zdrojového kódu umožňuje priame použitie knižnice Code Toolkit, ktorej vytvorenie je súbežným cieľom tejto práce.

Okrem inštrumentácie zdrojového kódu by bolo v prostredí .NET Framework tiež možné inštrumentáciu realizovať na preloženom kóde CIL (Common Intermediate Language tiež známeho ako Microsoft Intermediate Language – MSIL). Všetky .NET jazyky sa vždy najprv prekladajú do CIL a až potom (väčšinou pri prvom spustení) sa kód z CIL preloží do natívnych inštrukcií procesora[5]. Inštrumentácia CIL má výhodu v nezávislosti na konkrétnom jazyku, avšak inštrumentácia vo vybranom jazyku poskytuje jednoduchšiu korešpondenciu nameraných dát s jednotlivými príkazmi jazyka. Hoci je možné spätne zisťovať, ktoré inštrukcie CIL odpovedajú ktorým príkazom pôvodného jazyka pomocou ladiacich informácií v súbore typu .pdb[4] (v prostredí Mono[7] sa používa typ .mdb) a v prekladači jazyka je pritom možné zapnúť generovanie

tohto súboru spolu s CIL kódom[4], takéto zisťovanie korešpondencie s pôvodným jazykom, ale môže byť značne prácnejšie.

Na druhej strane je treba priznať, že vytvorenie reprezentácie CIL kódu a jeho modifikácia je podstatne jednoduchšie ako parsovanie C# programu. Navyše už existuje voľne dostupná knižnica Cecil[8], ktorá takúto prácu s CIL poskytuje. Inštrumentácia by tak mohla byť významne jednoduchšia. Je možné, že by bola pôvodne zvolená inštrumentáciu CIL, keby som si bol vedomí tejto výhody už v začiatočných fázach vývoja. Predtým som sa však nadchol pre myšlienku vytvorenia univerzálnej podpornej knižnice pre prácu so zdrojovým C# kódom a už som výskumu CIL nevenoval väčšiu pozornosť[1]. V každom prípade, i keď možno nebolo zvolené najjednoduchšie riešenie samotného profilovania, ako budúci úžitok knižnice Code Toolkit by mohol priniesť širšiu hodnotu než ponúka oblasť profilovania.

Alternatívnou možnosťou voči metóde inštrumentácie je využitie Microsoft .NET Profiling API. Medzi jeho výhody určite patrí možnosť sledovania všetkých udalostí automatickej správy pamäte ako je alokácia a odstránenie objektu, či začiatok kolekcie GC[2]. Tieto údaje nie je možné získať samotnou inštrumentáciou. Toto riešenie ale so sebou prináša nutnosť implementácie COM[32] servera, čo je jednak miernou komplikáciu samou o sebe[2], ale zároveň aj znamená úplné dištancovanie sa od možnej prenositeľnosti. Dielo bakalárskej práce síce nebolo písane s cieľom okamžitej prenositeľnosti napríklad

[1] Knižnicu Cecil som objavil až v pokročilej fáze projektu, keď som zisťoval ako by bolo možné pomocou CIL ukladať odkazy na lokálne premenné v zásobníku za účelom sledovania ich hodnoty v ľubovoľnom čase profilerom. Táto funkcia však nakoniec v tejto verzii profilera nebola použitá.

[2] Obzvlášť vzhľadom na fakt, že nemám s touto technológiu žiadne skúsenosti a v súčasnej dobe vzhľadom na jej perspektívy ani nepovažujem jej zvládnutie za obzvlášť prospešné.

na platformu Mono, avšak použité technológie boli vyberané tak, aby bol v budúcnosti možný nenáročný prenos. Užívateľské prostredie založené na WPF[4] predstavuje v tomto ohľade výnimku, ale vzhľadom na povahu práce predstavuje UI len menej významnú a relatívne jednoducho nahraditeľnú časť. Na záver diskusie o použití Microsoft .NET Profiling API treba ešte dodať, že z princípu akým spôsobom sa údaje získavajú, nie je možné uskutočňovať merania na úrovni príkazov (len na úrovni rutín), pretože profilovanie sa uskutočňuje čisto na základe spracovania volaní (udalostí) prichádzajúcich z CLR, ktoré však nevznikajú pri každom príkaze, ale pri vytvorení/zaniknutí objektu, začatí a skončení rutiny, či začatí kolekcie GC a pri ďalších významných udalostiach[4].

2.2 Čo všetko je možné merať

Profilovanie je v programovaní veľmi všeobecný pojem. Najčastejšie sa s ním spája meranie času trvania pozorovanej časti programu, či rôzne spôsoby merania spotrebovanej pamäte[9]. Tým ale zoznam zďaleka nekončí. Merať je tiež možné počet vykonaní jednotlivých príkazov, počet inštancii objektov jednotlivých tried programu, alebo vyťaženosť pevného disku, sieťových tokov, či spotreba iných prostriedkov operačného systému. To všetko sú vhodnými kandidátmi na meranie profilerom. Pre rozsiahlosť tejto oblasti sa práca snaží zamerať len na veličiny, ktoré sú najviac späté s jazykom. Cieľom je teda meranie počtu vykonaní sledovaných príkazov a ich času trvania a tiež meranie spotreby pamäte. Takéto zadanie je však ešte stále príliš obecné. V nasledujúcich odstavcoch preto budú jednotlivé možnosti bližšie rozobrané.

Zrejme najjednoduchším je počítanie počtu vykonaní vybraných príkazov. Pritom sa jedná o užitočnú informáciu, ktorá sa používa ani nie tak pri hľadaní výkonnostných nedostatkov ako pri zisťovaní pokrytia kódu testami[10]. Výskyt príkazov, ktoré sa nespustili ani raz, totiž znamenajú neúplne pokrytie testami. Bohužiaľ je potrebné dodať, že 100% pokrytie príkazov nemusí ani zďaleka znamenať úplné otestovanie programu, pretože pri volaní každého príkazu sa program môže nachádzať v rozličných stavoch znamenajúcich rôzne efekty

(výsledky) jediného príkazu. Hoci by takýto detektor vykonaných príkazov mohol byť praktickým samostatným nástrojom pri testovaní, aj pri profilovaní je možné jeho výsledky užitočne interpretovať. Ak mal napríklad nejaký meraný príkaz veľký počet opakovaní, znamená to, že nameraná priemerná hodnota lepšie vystihuje jeho priemerné chovanie ako keby sa vykonal (a teda i meral) len raz alebo malý počet krát.

Meranie časovej náročnosti sa bežne môže uskutočňovať na dvoch úrovniach. Hrubší pohľad sa dosiahne pri meraní trvania rutín (metód) ako celku, kým presnejšie informácie poskytuje meranie na úrovni jednotlivých príkazov. Táto práca sa snaží poskytnúť druhú presnejšiu metódu meraní, avšak z implementačných dôvodov bolo potrebné podstúpiť určité obmedzenie vo výbere príkazov. Jeden príkaz totiž môže obsahovať iné (napríklad príkaz cyklu obsahuje príkazy z tela cyklu) a pri meraní trvania vnoreného príkazu vzniká réžia so získaním a uložením nameraného údaju. Takže ak by bol súčasne meraný i nadriadený príkaz, k jeho trvaniu by sa nechcene pripočítala aj toto zdržanie merania jeho vnoreného príkazu. V budúcnosti by možno išlo toto obmedzenie odstrániť prípravným meraním trvania zdržania a odčítaním tejto hodnoty od nameraného času. Muselo by sa však štatisticky doložiť, že čas trvania merania a ukladania je dostatočne nemenný, aby mohli byť takto získané výsledky považované za dôveryhodné.

Pod pojmom profilovania pamäťovej spotreby sa dá predstaviť niekoľko typov meraní. Je možné napríklad počítať alokovanú pamäť inštanciami jednotlivých tried pre každú triedu samostatne. Ešte podrobnejšie je možné poskytnúť zoznam všetkých jednotlivých objektov s ich veľkosťami. Určitú informáciu o spotrebe pamäte by tiež poskytol len samotný počet inštancií jednotlivých tried. V predchádzajúcej podkapitole boli objasnené dôvody zvoleného spôsobu profilovania, ktorým je inštrumentácia t.j. vkladanie meracích príkazov priamo do zdrojového kódu v programe. Touto metódou by bolo možné relatívne jednoducho sledovať vznik a zánik jednotlivých objektov infiltrovaním vhodných príkazov do konštruktora a finalizéra[2]. Avšak takto by

sa dali sledovať len objekty tried deklarovaných v profilovanom zdrojovom kóde, teda nie napríklad objekty tried štandardných knižníc[4]. O niečo zložitejším by bolo vkladanie zaznamenávajúcich príkazov pred každým volaním `new`, čím by bolo možné sledovať vytvorenie každého objektu avšak opäť len, ak bol daný objekt vytvorený v profilovanom programe.

Na získanie prehľadu o existujúcich inštanciách všetkých tried sa teda ukazuje len jedna možnosť[3] a to začatím od všetkých GC koreňov[2] a postupným prechádzaním celým grafom živých objektov. Všetky aktívne objekty v pamäti totiž tvoria orientovaný graf, v ktorom vedie hrana z objektu A do objektu B práve vtedy, keď objekt A obsahuje referenciu na objekt B. Medzi GC korene patria všetky statické položky tried a tiež lokálne premenné a parametre metód zo zásobníka volania[4][2]. Z týchto koreňov je možné prechádzaním po hranách grafu dosiahnuť každý žijúci objekt a tak napríklad spočítať postupne ich veľkosti a počty inštancií pre jednotlivé typy.

Táto metóda je veľmi silným nástrojom umožňujúcim i zistenie veľkosti všetkých objektov dosiahnuteľných z daného GC koreňa v zdrojovom kóde (napríklad lokálna premenná) a ešte som sa nestretol s profilerom, ktorý by takúto informáciu poskytoval, hoci by mohla byť zaujímavá. Pôvodne mal profiler v tejto práci práve využívať možnosti tejto metódy, bohužiaľ však pre komplikovanosť jej realizácie a nutnosť riešenia i ďalších problémov bola jej realizácia odložená a nahradená významne jednoduchším meraním celkovej skonzumovanej pamäte pred a po vykonaní príkazu. Ako doplnok k nemu bolo

---

[3] Mimo použitia Microsoft .NET Profiling API, ktorého použitie bolo diskutované v predchádzajúcej podkapitole.

[4] Medzi GC korene patria i objekty, na ktoré ukazujú registre CPU[4], avšak v bodoch medzi príkazmi C# stačí uvažovať len vymenované GC korene. Vnútro príkazu môže byť preložené do CIL inštrukcií obsahujúcich ďalšie GC korene ako sú spomenuté registre, ktoré však majú zmysel len v rámci daného C# príkazu.

pridané i meranie počtu vykonaní kolekcií jednotlivých generácií pred a po príkaze.

2.3 Ďalšie súvisiace problémy

Pre správnu funkciu aplikácie profilera nestačí len údaje merať a ukladať. Tieto údaje je tiež potrebné preniesť do užívateľského prostredia a zobraziť. Takže je potrebné zabezpečiť komunikáciu medzi aplikáciou profilovaného programu a aplikáciou profilera. Keďže tieto aplikácie bežia v rôznych procesoch[5], bolo potrebné použiť medzi procesorovú komunikáciu (ďalej IPC – Inter-process communication)[3]. Riešenie komunikácie s profilovaným programom som však zaradil do knižnice Code Toolkit, keďže sa zrejme bude hodiť i pre iné nástroje. Popis jej realizácie je teda zahrnutý v časti 5.2.2 rozoberajúcej sa touto funkcionalitu knižnice.

Namerané dáta sú zapisované profilovaným programom a čítané asynchrónne (na požiadavok užívateľa), aby boli prenesené do aplikácie profilera, kde sa zobrazia. Tento asynchrónny prístup predstavuje nebezpečenstvo načítania nekonzistentných hodnôt a preto je nutné sa zaoberať problémom synchronizácie prístupu dát. Prvým riešením sa zdá byť použitie synchronizačných zámkov[11], avšak takéto riešenie by mohlo značne spomaliť zapisovanie nameraných údajov. A vzhľadom na to, že meranie sa môže potenciálne vyskytnúť pri každom príkaze, neustále uzamykanie by mohlo spôsobiť príliš značné spomalenie. Namiesto toho sa synchronizácia realizuje zastavením behu profilovaného programu na bezpečnom mieste t.j. mimo príkaz zapisujúci nameraný údaj. Zrejme nestačí jednoduché pozastavenie vlákna[4], keďže by sa vlákno mohlo zastaviť uprostred nevhodnej zapisovacej

[5] Spúšťanie profilovaného programu v tom istom procese profilera, ale len v druhej aplikačnej doméne[5] som zamietol, pretože by profilovaný program zdieľal GC[2] haldu s profilerom, čo by významne skresľovalo nameranú spotrebovanú pamäť.

inštrukcie. Na pozastavenie sa preto používa špeciálna booleanovská riadiaca premenná – príznak požiadavku na pozastavenie. Keď profiler potrebuje načítať namerané údaje, nastaví tento príznak a čaká až skutočne k pozastaveniu dôjde, čo pozná iným príznakom, ktorý signalizuje dosiahnutie pozastavenie.

Profilovaná aplikácia má na kritických miestach infiltrovaný malý kúsok kódu, ktorý len testuje nastavenie príznaku a ak zistí, že je nastavený, zaháji pozastavenie. Ak sa pozastavenie nepožaduje, tento malý kúsok kódu (controlling snippet) urobí len jediné porovnanie `if` na booleanovskú premennú, čo predstavuje prakticky zanedbateľné zdržanie. Príznak sa síce nastavuje asynchrónne, avšak nie je potrebné uzamkýnanie pred jeho prístupom, pretože v jednom vlákne sa do príznaku len zapisuje a v druhom sa len číta booleanovská hodnota, takže príznak nemôže obsahovať žiadnu nekonzistentnú hodnotu. Navyše, aby bola hodnota testovaného príznaku vždy aktuálna, v deklarácii príznaku bol použitý modifikátor `volatile`[12] zabraňujúci optimalizáciám, ktoré by mohli spôsobiť, že vlákno nepracuje s aktuálnou skutočnou hodnotou príznaku, ale vidí len hodnotu uloženú v cache pamäti procesora. Realizácia tohto mechanizmu pozastavenia bola tiež presunutá do knižnice Code Toolkit, pretože by mohla byť užitočná i pre iné programy. Podrobnejšiemu popisu jej implementácie sa venuje časť 5.2.3.

2.4 Zhrnutie

Po zvážení niekoľkých možností som sa nakoniec rozhodol uskutočňovať merania pomocou vkladania meracích inštrukcii (inštrumentácie) do zdrojového programu. Táto voľba bola ovplyvnená i snahou o vytvorenie univerzálnej knižnice poskytujúcej jednoduché načítanie a manipuláciu so zdrojovým kódom v jazyku C#. Knižnica tak môže umožniť jednoduché vkladanie meracích inštrukcií pred a za profilovaný príkaz. Spotreba príkazu (časová i pamäťová) sa potom získa ako rozdiel uplynutého času resp. spotrebovanej pamäte pred a po vykonaní príkazu.

Okrem samotného merania je potrebné vyriešiť aj ukladanie a prenášanie nameraných dát z profilovanej aplikácie do aplikácie profilera. Pre udržanie konzistentnosti je prístup k ním synchronizovaný pomocou pozastavovania v tzv. bezpečných bodoch. Toto pozastavenie aj s IPC komunikáciou je implementované vo vyššie zmienenej knižnici Code Toolkit. Knižnica Code Toolkit by sa tak mala stať silným nástrojom umožňujúcim nielen pohodlnú manipuláciu s kódom, ale i komunikáciu s ním za behu.

3. Existujúce profilery

Existuje množstvo profilerov pre jazyk C# od jednoduchších zameraných na určitý typ problému až po komplexné prostredia s množstvom funkcii. Na vytvorenie prehľadu o tom, čo je v súčasnosti k dispozícii je v tejto kapitole predstavených niekoľko nástrojov. Čo najobjektívnejší výber som sa snažil dosiahnuť použitím najrelevantnejších výsledkov internetového vyhľadávača[13], hoci by sa zrejme našlo niekoľko ďalších profilerov.

Nemálo profilerov umožňuje sledovať unmanaged[2] pamäť, využívanie rôznych prostriedkov (napr. prístupy k disku, sieti, atď.), profilovať ASP.NET[5] aplikácie, či služby Windows[5], alebo poskytujú mnohé ďalšie možnosti. Avšak táto práca sa zameriava len na profilovanie časového výkonu a používania managed[2] pamäte (haldy) a to len v bežných[6] programoch v jazyku C#, preto je výklad obmedzený len na vlastnosti týkajúceho sa tohto zamerania. Navyše zoznam vlastností u jednotlivých programoch nemusí byť kompletných ani vzhľadom na zameranie tejto práce. Pokúsil som sa vybrať vlastnosti, ktoré som subjektívne považoval za zaujímavé (užitočné, či originálne). Nasledujúce popisy preto nie sú vhodné ako prehľad možností jednotlivých nástrojov, ale skôr ako prehľad existujúcich možností profilovania ako takých.

3.1 Popis vybraných profilerov

Prof-it 1.02[14]

Prof-it je voľne dostupný profiler s licenciou GNU GPL[24]. Avšak jeho vývoj sa bohužiaľ zrejme zastavil v roku 2004. Meria len počet vykonaní jednotlivých príkazov („hit count"). Príkazy s „hit count" v danom intervale sa

[6] Bežnými programami sa myslí desktopová aplikácia t.j. konzolová aplikácia alebo aplikácia systému Windows.

dajú farebne zvýrazniť. Takto je možné definovať rôzne intervaly „hit count" a k ním voliť farby zvýraznenia.

NProfiler 0.1[15]

NProfiler je opäť zadarmo dostupný program s licenciou GNU GPL. Jeho vývoj za tiež bohužiaľ zrejme zastavil v roku 2002. Tento nástroj už poskytuje strom volaní jednotlivých metód, u ktorých sa zobrazuje čas ich celkového trvania.

ProfileSharp 1.3[16]

Ďalší voľne dostupný program je ProfileSharp. Okrem hierarchie volania s časmi trvania jednotlivých metód sú k dispozícii informácie o pamäti. Nástroj vypisuje jednotlivé inštancie tried s ich veľkosťou. U objektov je ešte možné zisťovať, aké z nich vedú referencie na druhé objekty.

CLR Profiler 2.0[17]

Posledný predstavený voľne dostupný profiler je úzko špecializovaný na sledovanie managed haldy. Dokáže zobraziť množstvo informácií o stave haldy počas behu profilovaného programu. K dispozícii je histogram veľkostí objektov i histogram dĺžky života jednotlivých objektov. Je možné zisťovať, ktoré objekty boli v hlade premiestňované pri kompaktizácii haldy[7]. Programom je ďalej možné zisťovať, ktoré objekty sú momentálne na ktorej adrese, či zobrazovať graf objektov prepojených vzájomnými referenciami. Život objektov je tiež možné sledovať na časovej osi, v ktorej sú vyznačené časy kolekcií.

[7] Automatická správa pamäte CLR po volaní kolekcie GC (garbage collection) presúva objekty na nižšie adresy tak, aby nevznikali pamäťové diery medzi objektmi. Tento proces sa nazýva kompaktizácia haldy resp. heap compacting.

Nasledujúce programy už nie sú zadarmo dostupné, ale jedná sa o prepracovaná prostredia s cenou v rádoch stoviek dolárov. Mnohé z nich poskytujú aj rozšírenie do rôznych verzií vývojového prostredia Microsoft Visual Studio[23].

YourKit Profiler for .NET 3.0.5[18]

Tento nástroj prináša graf vyťaženia procesoru profilovaným programom, ktorý sa priebežne automaticky aktualizuje (takýto graf bude ďalej označovaný ako online graf). Na výber sú dva režimy. V režime vzorkovania („sampling") sú merané len časy trvania rutín, kým v režime trasovania („tracing") sa počíta i počet volaní rutín. Časy metód sú k dispozícii jednak vrátane dcérskych volaní (volania vychádzajúce z danej rutiny) i ako vlastný čas rutiny bez dcérskych volaní.

Na profilovanie pamäte slúži online graf celkového využívania managed[2] haldy, v ktorom sú vyznačované aj časové úseky prebiehania kolekcie nepotrebných objektov (ďalej bude použitý názvov GC ako skratka pre garbage collection[2]). V profiler je tiež možné zisťovať počet živých inštancií jednotlivých tried a ich celkovú veľkosť (v bytoch).

.NET Memory Profiler 3.1[19]

.NET Memory Profiler poskytuje 3 pohľady. Prvým je zoznam tried, ktorý u každej triedy zobrazuje počet vytvorených/zrušených inštancií, celkovú alokovanú pamäť inštanciami a pre triedy implementujúce rozhranie `IDisposable`[4] i počet inštancii so zavolanou/nezavolanou metódou `Dispose()`. V druhom pohľade – zoznam inštancií vybranej triedy – je u jednotlivých objektov možné zisťovať ich generáciu[2], zoznam referencii idúcich z/do daného objektu, zásobník volaní v čase vytvorenia, zoznam ciest od koreňového objektu, hodnoty položiek objektu a tiež veľkosť vrátane dcérskych objektov (t.j. vrátane objektov, na ktoré vedie referencia z daného objektu) i veľkosť bez dcérskych objektov. Tretím pohľadom je zoznam

jednotlivých zásobníkov volania, pričom sa u každého zásobníka volania zobrazí zoznam objektov, ktoré boli v príslušnej rutine vytvorené.

JetBrains dotTrace 3.1[20]

Základom sledovania výkonu u tohto nástroja je strom volaní pozostávajúci z jednotlivých metód. Uzol metódy A je nadradením uzlom inej metódy B práve vtedy, keď metóda A volá metódu B. U metód je zobrazovaný čas trvania (celkový s/bez dcérskych volaní, minimálny, maximálny a priemerný) a počet vykonaní. U všetkých predchádzajúcich programov vrátane tohto sa profilované informácie neaktualizujú priebežne (online), ale zaktualizujú sa až po vytvorení snímku. Zaujímavosťou JetBrains dotTrace je možnosť porovnania dvoch výkonnostných snímok, takže je možné vidieť o koľko sa jednotlivé metódy zrýchlili/spomalili, resp. ako sa zmenil počet ich volaní. Porovnávanie snímok funguje aj pre pamäť, teda je možné zistiť koľko objektov daného typu vzniklo, či zaniklo v čase medzi dvoma vytvorenými snímkami. Medzi vlastnosť, ktorou nedisponovali predchádzajúce profilery, ešte patrí zobrazovanie počtu a veľkosti tzv. držaných („held") objektov jednotlivých tried, kde objekt B je držaným objektom objektu A, práve vtedy, keď na objekt B vedú práve len referencie z objektu A.

AQTime 5.42[21]

Originálnym prvkom profilera AQTime je možnosť zadávania „triggerov". „Trigger" dokáže zapnúť alebo vypnúť profilovanie pri zadanej udalosti. Ako udalosť je možné zadať spustenie vybranej metódy. V zozname rutín sa nachádza okrem už štandardných informácii i čas prvého spustenia a počet vyvolaných výnimiek. K dispozícii je aj pohľad na zdrojový kód, kde sú u metód na začiatku v komentári uvedené niektoré nameraných hodnoty. Novinkou je aj grafické znázornenie volaní v podobe grafu, kde uzly predstavujú rutiny a hrana reprezentujú volania medzi nimi, pričom u každej hrany je uvedený číslo – počet volaní. Možnosti profilovania pamäte sú podobné ako v iných vyššie uvedených programoch. Nespomenutá zostala ešte možnosť manuálneho zavolania GC v ľubovoľnom okamžiku počas behu profilovaného programu.

ANTS Profiler 3.2[22]

Program ANTS Profiler ponúka dva režimy: profilovanie na úrovni metód a pomalšie profilovanie na úrovni riadkov kódu. Rýchly prehľad sa snaží poskytnúť zobrazovaním súhrnných informácií obsahujúcich 10 najpomalších metód, 10 najpomalších riadkov, 10 najväčším žijúcich inštancií a 10 tried s najväčším počtom inštancií. Mnohé z jeho vlastností už boli spomínané v rôznych predchádzajúcich profileroch. Po výbere metódy v zozname sa okrem iných informácii zobrazí aj jej zdrojový kód, pričom priamo na každom riadku sa zobrazuje i čas trvania príslušného príkazu. Významné sprehľadnenie prináša možnosť filtrovania (podľa definovateľných pravidiel) a zoskupovania položiek zoznamu (položky – rutiny, triedy, či objekty - sa usporiadajú do podzoznamov s rovnakou hodnotou vybranej vlastnosti).

Performance Profiler v programe Visual Studio Team System 2008[23]

Profiler v prostredí Visual Studio navyše umožňuje v stromovej štruktúre volania metód otvoriť kritickú cestu vedúcu z vybranej metódy („hot path"), takže užívateľovi sa ukáže cesta volaní s najdlhším trvaním a metóda, ktorá toto spomalenia spôsobuje, je zvlášť označená. Množstvo položiek zoznamov je možné redukovať použitím dvoch špeciálnych filtrov. „Folding" zoskupuje položky, ktorých rozdiel hodnôt spotrebovaného času/pamäte je pod zvolenou prahovou hodnotou, kým „Trimming" položky s hodnotou pod prahom nezobrazuje vôbec. Mnohé ďalšie funkcie tohto nástroja už boli popísané v predchádzajúcich programoch.

3.2 Zhrnutie

Ako je podľa vyššie uvedených popisov vybraný profilerov vidieť, existuje už niekoľko programov na profilovanie času trvania vybraných riadkov alebo celých metód, či meranie počtu vytvorených objektov a ich pamäti, ktorú spotrebujú. Niektoré z nich sú veľmi komplexné nástroje poskytujúce množstvo ďalších informácií ako napr. graf volaní funkcií. Tieto programy sú však komerčné a ich cena sa nezriedka pohybuje v stovkách dolárov. Profiler

vytvorený v tejto práci síce neponúka až takú bohatú funkcionalitu, ale stále umožňuje získať podstatné informácie o výkonnosti programu v jednoduchom a prehľadnom užívateľskom rozhraní.

I keď existujú i voľne dostupné profilery, žiadny s ktorým som sa stretol, však na úrovni jednotlivých príkazov neumožňuje uskutočňovať ani meranie času. To je prednosťou vytvoreného môjho profilera na poli voľne dostupných programov. Navyše však umožňuje merať celkovú spotrebu pamäte jednotlivých príkazov v kilobytoch, s čím som sa nestretol ani u komerčných programov, keďže tieto spravidla merajú spotrebovanú pamäť ako celkový počet a veľkosť vytvorených objektov bez ohľadu na príkaz, v ktorom vznikli. Mnoho profilerov ponúka nejakú originálnu možnosť merania a profiler vytvorený v tejto práci má teda tiež čo ponúknuť ako doplnok k existujúcim programom.

4. Použitie programu C# Profiler

Aplikáciu C# Profiler je možné použiť na profilovanie jednovláknových C# 2.0 programov. Profilované programy sa otvárajú ako C# projekty vývojového prostredia Microsoft Visual Studio[23]. Bohužiaľ však nie je možné otvárať ľubovoľné C# projekty. Súčasná verzia knižnice Code Toolkit totiž zatiaľ nepodporuje kompiláciu projektov, ktoré obsahujú resources a tiež ktoré obsahujú odkazy na iné projekty. Odkazy na assembly sú však v projekte podporované.

V otvorenom projekte je možné v stromovej štruktúre vidieť hierarchiu tried (resp. štruktúr) a ich metód. Vo vybranej metóde je potom možné označovať príkazy, ktoré sa majú profilovať. Pri profilovaní času trvania príkazov nie je možné z technických dôvodov vybrať naraz príkazy tak, že jeden príkaz je vnorený v druhom, pretože pri meraní vnoreného príkazu sa pred neho a za ním vkladajú meracie inštrukcie, ktoré by znehodnotili výsledok merania vonkajšieho príkazu. Z rovnakého dôvodu je pri profilovaní času príkazov možné vybrať naraz len príkazy z jednej metódy, keďže druhá metóda by sa potenciálne mohla vykonať počas vykonávania príkazu z prvej metódy a jej príkazy by sa tak stali efektívne vnorenými príkazmi dotyčného príkazu z prvej metódy. Toto obmedzenie sa netýka profilovania pamäťovej spotreby, kde je možné príkazy voliť v ľubovoľných kombináciách a to z niekoľkých metód naraz.

Po spustení profilovanej aplikácie je možné v ľubovoľnom čase manuálne vyžiadať výsledky meraní, ktoré sa zobrazia pri označených príkazoch. Pri meraní času je možné sledovať priemerné a celkové trvanie príkazu ako aj minimálne a maximálne trvanie spomedzi všetkých vykonaní daného príkazu. Podobné informácie sa zobrazujú aj o spotrebovanej pamäti pri pamäťovom profilovaní. Navyše je však možné sledovať celkový počet GC kolekcií pre jednotlivé generácie, ktoré nastali počas vykonávania príslušného príkazu.

Podrobnejší popis užívateľského prostredia sa nachádza v prílohe D. V tejto kapitole je následne ešte uvedené doporučenie týkajúce sa plynulého pozastavenia a tiež ukážkové príklady použitia.

4.1 Pozastavovanie vo Windows Forms aplikáciách

V aplikáciách používajúcich Windows Forms[4] sa typicky v hlavnej metóde nachádza volanie podobné nasledujúcemu:

```
Application.Run(new Form1());
```

Programový kód 1

V takomto prípade nemôže program dosiahnuť bezpečný bod zastavenia, kým nie je vyvolaná udalosť. To sa prejaví často tým, že operácia zastavenia trvá až dovtedy, kým užívateľ napríklad neklikne na tlačítko profilovaného programu, ktoré spustí ošetrenie nejakej udalosti. Ak totiž kód ošetrujúci udalosť patrí zdrojovým kódom z otvoreného projektu, tak sa na začiatku tohto ošetrujúceho kódu nachádza bezpečné miesto na pozastavenie. Aby bolo možné sa tomuto nepríjemnému chovaniu s pozastavovaním vyhnúť, stačí to hlavnej metódy profilovaného programu tesne pred volaním `Application.Run` pridať nasledujúce príkazy:

```
System.Windows.Forms.Timer timer =
            new System.Windows.Forms.Timer();
timer.Interval = 500;
timer.Tick += new EventHandler(timer_Tick);
timer.Start();
```

Programový kód 2

A zároveň do tej istej triedy pridať prázdnu metódu:

```
static void timer_Tick(object sender, EventArgs e)
{
}
```

Programový kód 3

Tým sa každých 500 milisekúnd zavolá metóda `timer_Tick` a v jej tele už bude umiestnený bezpečný bod na zastavenie. Dôležité tiež je, že pozastavenie funguje správne len v jednovláknových aplikáciách. Našťastie sa udalosť tohto časovača spracováva pomocou fronty správ[4] a teda jeho ošetrujúca metóda beží v stále v tom istom vlákne aplikácie. Zároveň to i znamená, že počas pozastavenia prestáva okno reagovať na všetky udalosti.

4.2 Ukážkové použitia profilera

V tejto podkapitole je predstavených niekoľko prípadov použitia profilera na demonštratívnej aplikácii ProfilingSample, ktorej zdrojové kódy sa nachádza na priloženom CD-ROM médiu. Jedná sa o Windows Forms aplikáciu pozostávajúcu z jedného okna s tlačítkami spúšťajúcimi jednotlivé metódy použité na demonštráciu profilovania. K jednotlivým ukážkam je uvedená i interpretácia výsledkov, keďže namerané hodnoty nemusia vždy reprezentovať to, čo by sa na prvý pohľad mohlo zdať. Ukážky by teda mali poskytnúť i poučenie o možných problémoch správnej interpretácie meraní.

Pripomínam, že i do programu ProfilingSample bolo vložené pravidelné spúšťanie prázdnej metódy ako je uvedené v programových kódoch 2 a 3, aby sem mohol byť vložený bezpečný bod pozastavenia a pozastavenie tak mohlo byť realizované takmer okamžite.

Vzhľadom na to, že profiler umožňuje uskutočňovať merania času naraz len v jednej metóde, ukážky sú implementované tak, že všetky alternatívne prístupy riešenia rovnakého problému sú umiestnené do jednej metódy. V praxi však môže vzniknúť potreba súčasného porovnania efektívnosti kódov z rôznych

metód. V tomto prípade je potrebné štruktúru programu dočasne vhodne preusporiadať tak, aby sa časti kódu určené na porovnanie dostali do takejto jednej metódy, ak je to možné rozumným spôsobom dosiahnuť. Druhou možnosť je uskutočniť meranie dvakrát – samostatne pre obidve metódy a výsledky prvého merania si poznamenať na porovnanie s druhým meraním.

Namerané absolútne hodnoty sa pochopiteľne budú líšiť nielen medzi jednotlivými počítačmi, ale i pri za sebou nasledujúcich spusteniach. Avšak pre jednotlivé spustenia by sa od seba nemali líšiť príliš zásadne a hlavne pomer efektívnosti jednotlivých častí programu by mal byť relatívne stabilný. Pre presnejší odhad je pochopiteľne vhodné meranie opakovať a vychádzať z priemernej hodnoty.

4.2.1 Porovnanie efektívnosti rôznych algoritmov toho istého problému

Jednou z možností ako využívať aplikáciu CSharp Profiler je pomoc pri výbere efektívneho algoritmu riešiaceho určitý problém. Prvá ukážka predstavuje 3 rôzne spôsoby spočítania počtu prvočísel[25] v číselnom rozsahu od 1 do zvoleného N. V okne programu ProfilingSample je možné voliť číslo N v číselnom poli označenom „Max number" a tlačítkom „Count primes" potom spustiť všetky 3 algoritmy.

Na porovnanie efektívnosti všetkých 3 metód je vhodné v profilovaní CPU zvoliť všetky príkazy prvej úrovne v metóde `CountPrimes` z triedy `PrimesCounter` t.j. v metóde, kde sú všetky 3 algoritmy implementované. Príkazmi prvej úrovne sa myslia príkazy, ktoré nie sú vnorené v iných príkazoch. Principiálne by nebolo na škodu merať všetky príkazy všetkých 3 algoritmov, ale pre vyššie spomínané obmedzenie to profiler neumožňuje a na porovnanie efektivity je dostačujúce poznať čas vykonávania príkazov prvej úrovne. Nasledujúce merania boli vykonané pre počítanie počtu prvočísel do čísla 100 000.

Na použitom počítači[8] boli pre prvý algoritmus dosiahnuté nasledujúce hodnoty:

```
// 1.Approach - each number is tested for being prime
// where the testing uses trivial division test.
0,0 ms int count1 = 0;
4993,5 ms for (int i = 2; i <= max; i++) {
    bool isPrime = true;
    for (int j = 2; j < i; j++) {
        if (i % j == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime) { count1++; }
}
```

Obr. 1 – Výsledky meraní

Prvočísla boli v prvom algoritme hľadané postupným testovaním deliteľnosti od 2 do hodnoty o jedna menej ako skúmané číslo. Ako je vidieť z obr. 1 tento triviálny algoritmus trval takmer 5 sekúnd.

Druhý algoritmus testujúci deliteľnosť len do druhej odmocniny skúmaného čísla[25] dosiahol takmer 100 násobne lepší čas:

```
// 2.Approach - the same as previous one but division
// test is only up to square root of tested number.
0,0 ms int count2 = 0;
55,7 ms for (int i = 2; i <= max; i++) {
    bool isPrime = true;
    int sqrtOfI = (int)Math.Ceiling(Math.Sqrt(i));
    int maxTest = Math.Min(sqrtOfI, i - 1);
    for (int j = 2; j <= maxTest ; j++) {
        if (i % j == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime) { count2++; }
}
```

Obr. 2 – Výsledky meraní

[8] Notebook Intel Core 2 Duo 1,6 GHz, 2GB RAM s Microsoft Windows Vista Home Premium

Tento pomer je samozrejme tým väčší, čím je väčšie maximálne testované číslo. Zhruba ďalšie 10 násobné zlepšenie dosahuje 3. algoritmus, ktorý najprv nájde prvočísla metódou Eratostenového sita[26] a potom spočíta ich počet:

```
// 3.Approach - Primes are found using sieve
// of Eratosthenes and then are counted.
0,0 ms int count3 = 0;
0,0 ms bool[] primeSieve = new bool[max + 1];
0,6 ms for (int i = 2; i <= max; i++) {
    primeSieve[i] = true;
}

0,0 ms int sqrtOfMax = (int)Math.Ceiling(Math.Sqrt(max));
4,3 ms for (int i = 2; i <= sqrtOfMax; i++) {
    if (primeSieve[i]) {
        for (int j = 2; j <= max / i; j++) {
            primeSieve[i * j] = false;
        }
    }
}
1,0 ms foreach (bool isPrime in primeSieve) {
    if (isPrime) { count3++; }
}
```

Obr. 3 – Výsledky meraní

Získané výsledky nie sú vôbec prekvapujúce. Dalo sa očakávať, ktorý algoritmus bude najrýchlejší a ktorý najpomalší, ale navyše sme získali predstavu koľkonásobne sú jednotlivé algoritmy lepšie pre zvolené N. Toto číslo by sa zrejme dalo odhadnúť i matematicky podľa pomeru počtu vykonaných príkazov jednotlivých algoritmov. V jednotlivých algoritmoch sú však použité rôzne príkazy s rôznym trvaním. Navyše použitie profilera poskytuje jednoduchší spôsob získania tohto pomeru. Pre zaujímavosť si tiež môžeme pozrieť počty vykonaní príkazov vnútorných cyklov ako údaj „Hit count" po vybratí profilovania týchto príkazov a premiestnení kurzora myši nad požadovaný príkaz. Obr. 4 bol získaný pri počítaní počtu prvočísel do čísla N=10 000.

```
// 1.Approach - each number is tested for being prime
// where the testing uses trivial division test.
int count1 = 0;
for (int i = 2; i <= max; i++) {
    bool isPrime = true;
    for (int j = 2; j < i; j++) {
        0,0 ms if (i % j == 0) {
            is  Hit count: 5775223
            br   Sum time: 7825,7 ms
        }        Min time: 0,0 ms
    }            Max time: 3,5 ms
    if (isPrime) { count1++; }
}
```

Obr. 4 – Výsledky meraní

Ako je vidieť z obr.4 testovanie na deliteľnosť sa v 1.algoritme vykonalo 5 775 223 krát. Na tomto obrázku je možné si tiež všimnúť, že celkový čas všetkých vykonaní tohto jediného príkazu (takmer 8 sekúnd) je väčší ako celkový čas trvania celého algoritmu s N=100 000. Je to spôsobené tým, že príkaz sa vo väčšine prípadov skončí extrémne rýchlo a teda je pomer trvania meracej réžie a skutočného vykonávania príkazu nezanedbateľný. Navyše sa meranie mnohokrát opakuje a teda sa tento rozdiel od skutočného trvania významne akumuluje. Pri profilovaní extrémne rýchlych a mnohokrát sa opakujúcich príkazov preto treba mať túto nedokonalosť merania na pamäti. Pre porovnanie sú ešte zobrazené počty opakovaní príkazov vnútorných cyklov obidvoch ďalších algoritmov tiež pre N=10 000:

```
// 2.Approach - the same as previous one but division
// test is only up to square root of tested number.
int count2 = 0;
for (int i = 2; i <= max; i++) {
    bool isPrime = true;
    int sqrtOfI = (int)Math.Ceiling(Math.Sqrt(i));
    int maxTest = Math.Min(sqrtOfI, i - 1);
    for (int j = 2; j <= maxTest ; j++) {
        0,0 ms if (i % j == 0) {
            is  Hit count: 118755
            br   Sum time: 149,4 ms
        }        Min time: 0,0 ms
    }            Max time: 0,3 ms
    if (isPrime) { count2++; }
}
```

Obr. 5 (prvá časť) – Výsledky meraní

```
// 3.Approach - Primes are found using sieve
// of Eratosthenes and then are counted.
int count3 = 0;
bool[] primeSieve = new bool[max + 1];
for (int i = 2; i <= max; i++) {
    primeSieve[i] = true;
}

int sqrtOfMax = (int)Math.Ceiling(Math.Sqrt(max));
for (int i = 2; i <= sqrtOfMax; i++) {
    if (primeSieve[i]) {
        for (int j = 2; j <= max / i; j++) {
            0,0 ms primeSieve[i * j] = false;
        }
    }
}
foreach (bool isPr  Hit count: 17991
                    Sum time: 24,1 ms
                    Min time: 0,0 ms
                    Max time: 0,3 ms  eve) {
    if (isPrime) { count3++; }
}
```

Obr. 5 (pokračovanie) – Výsledky meraní

4.2.2 Skúmanie efektivity pomocných jazykových elementov

V tejto ukážke sa najprv snažím zistiť, aké zdržanie prináša používanie prístupu k položkám objektu pomocou vlastnosti[12]. Do položky objektu sa najprv zapisuje pomocou vlastnosti a potom priamo. Obidva spôsoby sú opakované a tento počet opakovaní je možné nastavovať v poli „Repeat count". Celý test je prevedený dvakrát – raz pre objekt triedy a druhýkrát pre objekt štruktúry (`struct`), aby sa dalo posúdiť, či má toto rozlíšenie vplyv na výkon. Obidva prístupy sú implementované v jednej metóde volanej po stlačení tlačítka „Test language features". Pre 1 000 000 opakovaní boli namerané nasledujúce hodnoty aké ukazuje obr. 6.

```
// 1.Approach - using property on classes
0,0 ms DataClass data1 = new DataClass();
9,0 ms for (int i = 0; i < repeatCount; i++) {
    data1.Property = i;
}

// 2.Approach - using field on classes
0,0 ms DataClass data2 = new DataClass();
3,8 ms for (int i = 0; i < repeatCount; i++) {
    data2.Field = i;
}

// 3.Approach - using property on structs
0,0 ms DataStruct data3 = new DataStruct();
9,8 ms for (int i = 0; i < repeatCount; i++) {
    data3.Property = i;
}

// 4.Approach - using field on structs
0,0 ms DataStruct data4 = new DataStruct();
3,8 ms for (int i = 0; i < repeatCount; i++) {
    data4.Field = i;
}

// 5.Approach - local variable just for comparison
0,0 ms int local;
3,7 ms for (int i = 0; i < repeatCount; i++) {
    local = i;
}
```

Obr. 6 – Výsledky meraní

Z týchto meraní vyplývajú tri pozorovania. Medzi objektmi tried a štruktúr nie je žiadny významný rozdiel z hľadiska času prístupu k položke a vlastnosti objektu. Toto pozorovanie je trochu neočakávané, pretože štruktúra ako hodnotový typ sa nachádza priamo na zásobníku, kým trieda ako referenčný typ obsahuje na zásobníku len odkaz na haldu[2] a teda na získanie skutočnej adresy hodnoty je treba najprv prečítať adresu zo zásobníka. Mohlo by to byť spôsobené buď tým, že prekladač optimalizuje prístup k objektu a po celú dobu cyklu si uchováva skutočnú adresu hodnoty v pomocnej premennej (a tá sa zrejme po preklade do natívneho kódu uchováva v registre procesora), alebo druhým vysvetlením by mohlo byť, že ostatné inštrukcie cyklu spotrebujú nepomerné množstvo času a tak je čas výpočtu adresy položky objektu celkom nepozorovateľný.

V každom prípade je ale jasne vidieť, že priamy prístup k položke je viac ako dvakrát rýchlejší než s použitím vlastnosti. Pri návrhu tried sa však dôrazne

doporučuje nevystavovať verejné položky, ale obaľovať ich vlastnosťami[27]. Toto doporučenie má svoje opodstatnenie a výsledky tohto pokusu by rozhodne nemali slúžiť ako argument proti tomuto doporučeniu, ale na druhej strane je dobré si byť vedomí toho, že v kritických častiach programu, kde sa prístup k dátam objektu extrémne opakuje a rýchlosť nie je dostatočná, je možné dosiahnuť významné zrýchlenie priamym prístupom k položkám. V takomto prípade je ale tiež dobré zvážiť, či by nebolo možné a vhodnejšie takúto kritickú operáciu vložiť ako metódu priamo do triedy objektu, aby sa položky triedy nemuseli verejne vystavovať.

Napokon tretie pozorovanie je možné vyčítať s posledného piateho prístupu, v ktorom sa nepristupuje k položke objektu ale len k lokálnej premennej. Ukazuje sa, že prístup k položke objektu je v cykle rovnako efektívny ako prístup k lokálnej premennej. Tento jav by sa dal znovu vysvetliť dvoma spôsobmi ako v prípade paradoxu v prvom pozorovaní, kde sa neukázal žiadny rozdiel medzi prístupom k položke objektu hodnotového a referenčného typu.

Inou jazykovou pomôckou o ktorej efektivitu sa môžeme zaujímať je cyklu `foreach`. V ďalšom pokuse prebieha iterácia cez pole (1 000 000 prvkové) a porovnávajú sa dva prístupy – cyklus `for` a `foreach`:

```
0,0 ms int x;
2,4 ms int[] array = new int[repeatCount];

// 1.Approach - 'for' loop
14,5 ms for (int i = 0; i < array.Length; i++) {
    x = array[i];
}

// 2.Approach - 'foreach' loop
15,9 ms foreach (int value in array) {
    x = value;
}

// 1.Approach again to see that results are probably affected by caching
10,9 ms for (int i = 0; i < array.Length; i++) {
    x = array[i];
}

// 2.Approach again
15,9 ms foreach (int value in array) {
    x = value;
}
```

Obr. 7 – Výsledky meraní

Z prvých dvoch meraní by sa mohlo zdať, že obidva prístupy trvajú približne 15 milisekúnd. Avšak po následnom zopakovaní sa ukazuje, že druhý priebeh `for` cyklu je o viac ako 1/3 rýchlejší. Pokus bol mnohokrát zopakovaný a toto pozorovanie sa zakaždým potvrdilo. Druhý beh `for` cyklu by mohol byť zakaždým rýchlejší možno vďaka cacheovaniu časti poľa. Avšak `foreach` cyklus má stabilne zhruba rovnaké trvanie ako prvý pomalší `for` cyklus. Tento rýchlostný rozdiel nie je taký obrovský, ale i tretinové zrýchlenie by pre kritickú časť kódu mohlo byť dostatočne prínosné.

4.2.3 Efektívnosť štandardných tried .NET Frameworku

Jednou z často používaných štandardných .NET Framework tried je `System.Collections.Generic.List<T>`[4]. V nasledujúcom pokuse bude porovnaná jej efektivita s obyčajným poľom. Najprv sa postupne naplní N prvkové pole a potom sa rovnakým počtom elementov naplní objekt triedy `System.Collections.Generic.List<T>`. Počet vložených elementov, N, je možné meniť v poli „Repeat count" programu ProfilingSample a metódu s experimentom je možné spustiť tlačítkom „Test .NET Collections". Výsledky meraní pre N=1 000 000 znázorňuje nasledujúci obrázok:

```
// 1.Approach - using array
1,9 ms int[] array = new int[repeatCount];
9,0 ms for (int i = 0; i < repeatCount; i++) {
    array[i] = i;
}

// 2.Approach - using list
0,0 ms List<int> list1 = new List<int>();
26,3 ms for (int i = 0; i < repeatCount; i++) {
    list1.Add(i);
}

// 3.Approach - using list with preserved capacity
0,9 ms List<int> list2 = new List<int>(repeatCount);
17,2 ms for (int i = 0; i < repeatCount; i++) {
    list2.Add(i);
}
```

Obr. 8 – Výsledky meraní

Ako bolo možné očakávať, napĺňanie objektu poľa je významne (asi 2,5 krát) rýchlejšie. Tretí prístup ukazuje, že napĺňanie zoznamu je možné zrýchliť rezervovaním potrebnej kapacity. V tomto príklade si pozrime ešte spotrebovanú pamäť jednotlivých príkazov:

```
// 1.Approach - using array
3906,3 kint[] array = new int[repeatCount];
0,0 kfor (int i = 0; i < repeatCount; i++) {
    0,0 karray[i] = i;
}

// 2.Approach - using list
0,0 kList<int> list1 = new List<int>();
5952,9 kfor (int i = 0; i < repeatCount; i++) {
    0,0 klist1.Add(i);
}

// 3.Approach - using list with preserved capacity
3914,3 kList<int> list2 = new List<int>(repeatCount);
0,0 kfor (int i = 0; i < repeatCount; i++) {
    0,0 klist2.Add(i);
}
```

Obr. 9 – Výsledky meraní

Pri vytvorení poľa sa spotrebuje približne 3 906,3 KB x 1024 ≅ 4 000 050 bytov. Teda celočíselný prvok poľa zaberá 4 byty, čo sa dalo očakávať. Pri postupnom napĺňaní prázdneho zoznamu sa spotrebovalo asi 5 952,9 kB, čo je o 20% viac ako je skutočná potrebná kapacita. V skutočnosti sa počas tohto cyklu alokovalo ešte viac pamäte, ale táto pamäť musela byť uvoľnená počas behu cyklu vďaka GC kolekciám ako ukazuje detail meraní po premiestnení kurzora myši nad príkaz cyklu:

```
// 2.Approach - using list
0,0 kList<int> list1 = new List<int>();
5952,9 kfor (int i = 0; i < repeatCount; i++) {
    0,0 k
}
    Hit count: 1
    Sum memory: 5952,9 k
    Min memory: 5952,9 k
    Max memory: 5952,9 k
    Sum gen.0: 2
    Sum gen.1: 2
    Sum gen.2: 2

// 3.App                ist with preserved capacity
3914,3 kL                = new List<int>(repeatCount);
0,0 kfor                 < repeatCount; i++) {
    0,0 k
}
```

Obr. 10 – Výsledky meraní

Tieto alokácie boli zrejme použité pri mnohonásobnom vytváraní väčšieho interného poľa pre ukladanie elementov.

Napokon si ešte porovnajme efektivitu sekvenčného čítania z poľa a zoznamu:

```
// 1.Approach - using array
11,5 ms for (int i = 0; i < repeatCount; i++) {
    x = array[i];
}

// 1.Approach again to consider relevancy of results
11,5 ms for (int i = 0; i < repeatCount; i++) {
    x = array[i];
}

// 2.Approach - using list
17,6 ms for (int i = 0; i < repeatCount; i++) {
    x = list1[i];
}

// 2.Approach again to consider relevancy of results
17,1 ms for (int i = 0; i < repeatCount; i++) {
    x = list1[i];
}

// 3.Approach - using list and 'foreach' loop
21,3 ms foreach (int value in list1) {
    x = value;
}
```

Obr. 11 – Výsledky meraní

Čítanie z poľa aj zo zoznamu bolo uskutočnené dvakrát za sebou, aby sa odhalili prípadné účinky cacheovania[9], na ktoré bolo poukázané v predchádzajúcej časti 4.5.2. Po opakovanom čítaní však k výraznému zrýchleniu nedošlo. Čítanie z poľa sa celkom pochopiteľne javí byť rýchlejšie ako čítanie zo zoznamu. Posledné meranie navyše opäť potvrdzuje, že `for` cyklus je o niečo rýchlejší ako `foreach` cyklus.

[9] I keď si nie som istý, že sa jedná o zrýchlenie v dôsledku cacheovania.

4.2.4 Efektívne generovanie xml dokumentu

Množstvo programov nie len na platforme .NET Framework pracuje s xml[33] formátom a potreba generovania xml nie je vôbec neobvyklá. Preto posledná ukážka použitia profilera predstavuje 5 rôznych spôsobov ako xml dokument generovať a poukazuje na ich efektívnosť čo do rýchlosti trvania i spotreby pamäte. Program ProfilingSample obsahuje metódu generujúcu xml dokument s nasledujúcou štruktúrou:

```
<?xml version='1.0' ?>
<Body>
    <Value>1</Value>
    <Value>2</Value>
    <Value>3</Value>
    . . .
</Body>
```

Počet značiek `<Value>` obsahujúcich číslo je možné nastavovať v poli „Repeat count". Generovanie sa spustí tlačítkom „Generate xml". Všetkých 5 spôsobov generovania je opäť umiestnených v jednej metóde, aby ich bolo možné naraz profilovať. Medzi jednotlivými generovaniami je však vložené explicitné volanie GC kolekcie, aby sa v jednotlivých generovaniach v GC halde nenachádzali nevyčistené objekty z predchádzajúcej činnosti. Tieto pozostatkové objekty by totiž mohli znehodnotiť meranie tým, že by predčasne spôsobili GC kolekciu počas generovania xml a tým ho spomalili a navyše i uvoľnili pamäť, čo by sa javilo ako celková menšia spotreba pamäte generovania. Všeobecne je preto pre zvýšenie objektívnosti vhodné explicitne volať GC kolekciu pred profilovaným kódom. Avšak toto platí, keď je potrebné merať efektívnosť izolovaného kódu. Naopak, ak je potrebné sledovať správanie určitého kódu v rámci programu ako celku, je takéto vynútené volanie GC kolekcie nežiadúce.

V tomto experimente bude generovaný xml dokument s vyššie uvedenou štruktúrou so značkami `<Value>` s číslami od 1 do 10 000. Prvý algoritmus generuje xml skladaním reťazcov. Pri jeho meraní boli zistené nasledujúce časy:

```
// 1.Approach - string concatenating
0,0 ms string xmlString = "<?xml version='1.0' ?>";
0,0 ms xmlString += "<Body>";
5536,8 ms for (int i = 0; i < itemsCount; i++) {
    xmlString += "<Value>" + i.ToString() + "</Value>";
}
0,6 ms xmlString += "</Body>";
```

Obr. 12 – Výsledky meraní

Tento spôsob trvajúci zhruba 5,5 sekundy zrejme nebude patriť medzi najrýchlejšie. Podstatne rýchlejšie je použitie.NET Framework triedy `System.Text.StringBuilder`[4]:

```
// 2.Approach - using StringBuilder
0,0 ms StringBuilder xmlBuilder = new StringBuilder();
0,0 ms xmlBuilder.Append("<?xml version='1.0' ?>");
0,0 ms xmlBuilder.Append("<Body>");
10,3 ms for (int i = 0; i < itemsCount; i++) {
    xmlBuilder.Append("<Value>" + i.ToString() + "</Value>");
}
0,0 ms xmlBuilder.Append("</Body>");
```

Obr. 13 – Výsledky meraní

Zrýchlenie je viac ako 500 násobné. Avšak aj v tomto druhom algoritme je ešte stále použité spájanie reťazcov pri tvorbe jednotlivých `<Value>` značiek. Pozrime sa preto na čas trvania algoritmu dôsledne používajúceho `StringBuilder` bez skladania reťazcov:

```
// 3.Approach - using StringBuilder more thoroughly
0,0 ms xmlBuilder = new StringBuilder();
0,0 ms xmlBuilder.Append("<?xml version='1.0' ?>");
0,0 ms xmlBuilder.Append("<Body>");
10,0 ms for (int i = 0; i < itemsCount; i++) {
    xmlBuilder.Append("<Value>");
    xmlBuilder.Append(i);
    xmlBuilder.Append("</Value>");
}
0,0 ms xmlBuilder.Append("</Body>");
```

Obr. 14 – Výsledky meraní

Zdá sa, že zrýchlenie je v tomto prípade zanedbateľné.

Sekvenčné generovanie xml dokumentu je možné dosiahnuť i pomocou .NET triedy `System.Xml.XmlWriter`[4]. Táto trieda je priamo určená k tomuto účelu, a teda by sa dalo očakávať, že jej použitie bude veľmi efektívne. Xml dokument môže byť generovaný do súboru, streamu (`System.IO.Stream`) alebo inštancie triedy `StringBuilder`. V poslednom prípade by zrejme výsledné trvanie malo byť podobné ako v treťom algoritme. Ako však ukazuje nasledujúce meranie, `XmlWriter` so zápisom do inštancie `StringBuilder` je o niečo pomalší ako bolo priame generovanie v `StringBuilder` inštancii:

```
// 4.Approach - using XmlWriter
0,0 ms StringBuilder xmlWriterStorage = new StringBuilder();
0,0 ms XmlWriter xmlWriter = XmlWriter.Create(xmlWriterStorage);
0,0 ms xmlWriter.WriteStartDocument();
0,0 ms xmlWriter.WriteStartElement("Body");
14,9 ms for (int i = 0; i < itemsCount; i++) {
    xmlWriter.WriteStartElement("Value");
    xmlWriter.WriteString(i.ToString());
    xmlWriter.WriteEndElement();
}
0,0 ms xmlWriter.WriteEndElement();
0,0 ms xmlWriter.Close();
```

Obr. 15 – Výsledky meraní

Napokon ešte existuje možnosť generovať xml dokument pomocou budovania objektového modelu xml v inštancii triedy `System.Xml.XmlDocument`[4] a následného uloženia tohto modelu do reťazca:

```
// 5.Approach - building an XmlDocument instance
0,0 ms XmlDocument xmlDocument = new XmlDocument();
0,0 ms XmlElement tagBody = xmlDocument.CreateElement("Body");
0,0 ms xmlDocument.AppendChild(tagBody);
17,6 ms for (int i = 0; i < itemsCount; i++) {
    XmlElement tagValue = xmlDocument.CreateElement("Value");
    XmlText number = xmlDocument.CreateTextNode(i.ToString());
    tagValue.AppendChild(number);
    tagBody.AppendChild(tagValue);
}
13,7 ms string xmlDocumentContent = xmlDocument.InnerXml;
```

Obr. 16 – Výsledky meraní

Ako je i podľa výsledkov meraní vidieť, tento algoritmus je viac ako dvojnásobne pomalší, pretože sa najprv musí vytvoriť objektový model xml a až potom dochádza ku generovaniu xml textu. Avšak je treba si uvedomiť, že tento prístup má oproti predchádzajúcim algoritmom veľkú výhodu, že umožňuje generovať xml dokument nesekvenčným spôsobom.

Porovnajme si jednotlivé prístupy znovu avšak tentoraz z hľadiska spotreby pamäte. Opäť bude generovaný xml dokument s 10 000 značkami `<Value>`. Jedna značka obsahuje priemerne okolo 19 znakov („`<Value>1234</Value>`“), keďže najviac sa v nej bude vyskytovať 4 ciferných čísel. Znaky sú ukladané v kóde Unicode[4], čiže jeden znak bude zaberať 2 byty. Hotový xml dokument by teda mal spolu zaberať približne 10 000 x 19 x 2 = 380 000 bytov. Zdá sa teda, že prvý algoritmus nespotrebuje relatívne o moc viac pamäte ako je nutné:

```
// 1.Approach - string concatenating
0,0 k string xmlString = "<?xml version='1.0' ?>";
8,0 k xmlString += "<Body>";
693,8 k for (int i = 0; i < itemsCount; i++) {
    169,7 k xmlString += "<Value>" + i.ToString() + "</Value>";
}
369,0 k xmlString += "</Body>";
```

Obr. 17 – Výsledky meraní

Reťazce sú ale immutable[34] a teda každé spojenie dvoch reťazcov spôsobilo vznik nového reťazca. Preto v skutočnosti v tomto cykle vzniklo obrovské množstvo reťazcov, ktoré boli následne vyčistené v mnohých GC kolekciách:

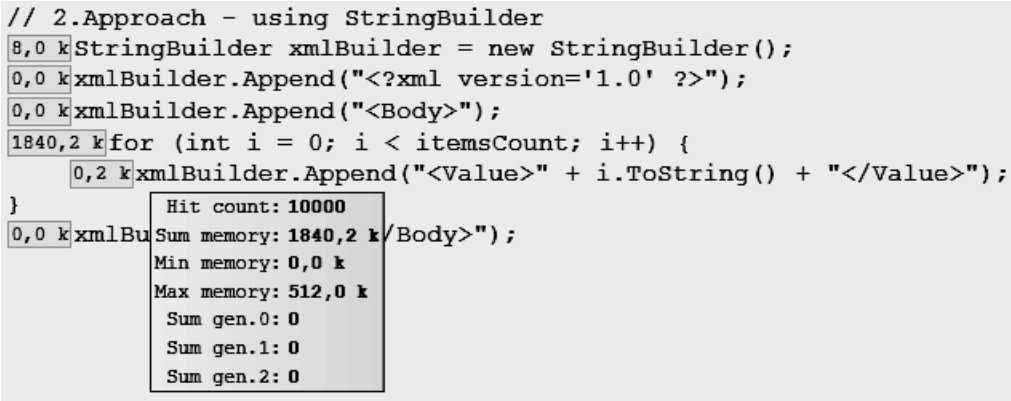
```
// 1.Approach - string concatenating
0,0 k string xmlString = "<?xml version='1.0' ?>";
8,0 k xmlString += "<Body>";
693,8 k for (int i = 0; i < itemsCount; i++) {
    169,7 k xmlString += "<Value>" + i.ToString() + "</Value>";
}
369,0 k xmlSt
0,0 k GC.Col
```

```
Hit count: 10000
Sum memory: 1697123,0 k
Min memory: 0,0 k
Max memory: 376,7 k
Sum gen.0: 628
Sum gen.1: 550
Sum gen.2: 545
```

Obr. 18 – Výsledky meraní

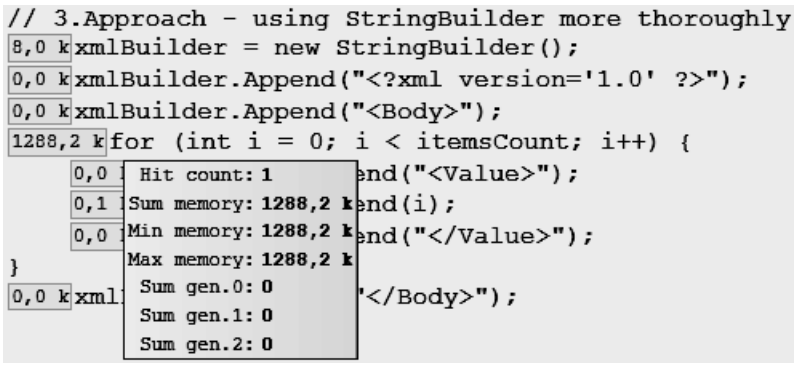
Takže na vysvetlenie hodnota 693,8 KB pri cykle znamená, že rozdiel použitej pamäte pred a po cykle je približne 700 KB, ale vnútri cyklu bolo spolu alokovaných takmer 1,7 GB pamäte, hoci tieto alokácie boli priebežne čistené v mnohých GC kolekciách. A to je aj dôvod prečo je tento spôsob tak extrémne pomalý.

Naproti tomu v druhom algoritme nedošlo k žiadnym GC kolekciám, avšak po jeho dokončení muselo v pamäti zostať asi 1 MB nevyčistených objektov:



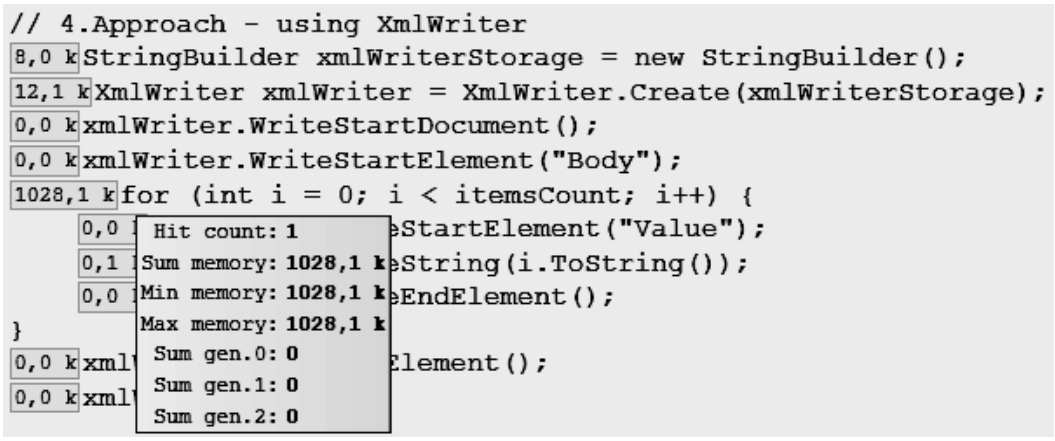
Obr. 19 – Výsledky meraní

V treťom algoritme tiež nedošlo ku GC kolekcii a navyše po ňom zostalo zhruba o 550 KB menej nevyčistených objektov zrejme kvôli úplnému odstráneniu spájania reťazcov:



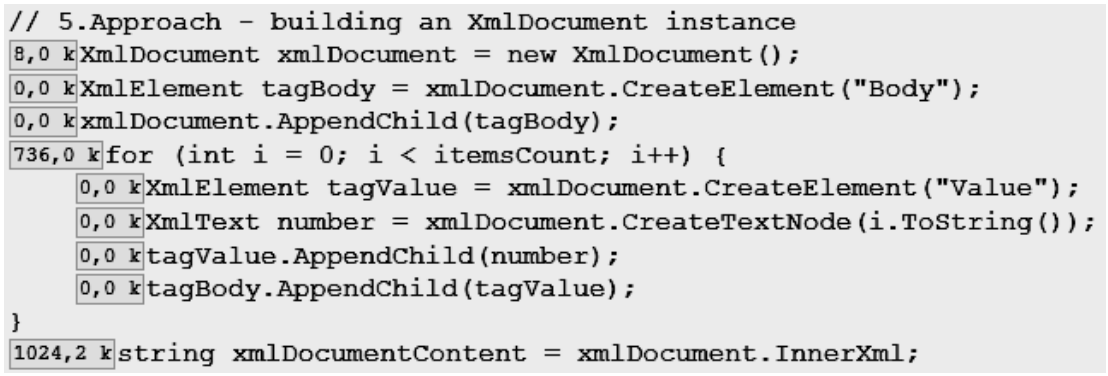
Obr. 20 – Výsledky meraní

Aj 4.algoritmus je na tom s pamäťovou spotrebou veľmi podobne:



Obr. 21 – Výsledky meraní

V poslednom 5. algoritme sa najprv asi 736 KB spotrebuje na vybudovanie objektového modelu xml a následne sa použije ďalší približne 1 MB na vytvorenie xml reťazca:



Obr. 22 – Výsledky meraní

V tomto prípade tiež nedošlo k žiadnej GC kolekcii.

Na záver ešte je možné si všimnúť, že pri všetkých 3 alokáciách prázdnej inštancie `StringBuilder` bola spotrebovaná pamäť 8 KB. Mohlo by sa zdať, že prázdny objekt `StringBuilder` zrejme vždy takúto pamäť zaberá. Na overenie som preto na konci metódy s generátormi xml umiestnil vytvorenie ďalšieho objektu `StringBuilder`. Ako ukazuje obr. 23 žiadnych 8KB nie je alokovaných.

```
// Create a StringBuilder instance to see how much memory will be used
0,0 k StringBuilder sb = new StringBuilder();
0,0 k sb.Append("test");
```

Obr. 23 – Výsledky meraní

Lepšie vysvetlenie je teda také, že po volaní GC kolekcie je prvá alokácia rozšírená na 8KB a zhodou okolností sa takéto volanie GC kolekcie nachádza pred všetkými 3 vytvoreniami objektov typu `StringBuilder`. Toto vysvetlenie navyše potvrdzuje spotrebovanie 8KB v alokáciách aj po ďalších 2 volaniach GC kolekcie – v 1. algoritme pri prvom spájaní reťazcov a v 5.algoritme pri vytváraní prázdneho objektu triedy `XmlDocument`. Hneď na začiatku 1.algoritmu sa nachádza priradenie reťazca:

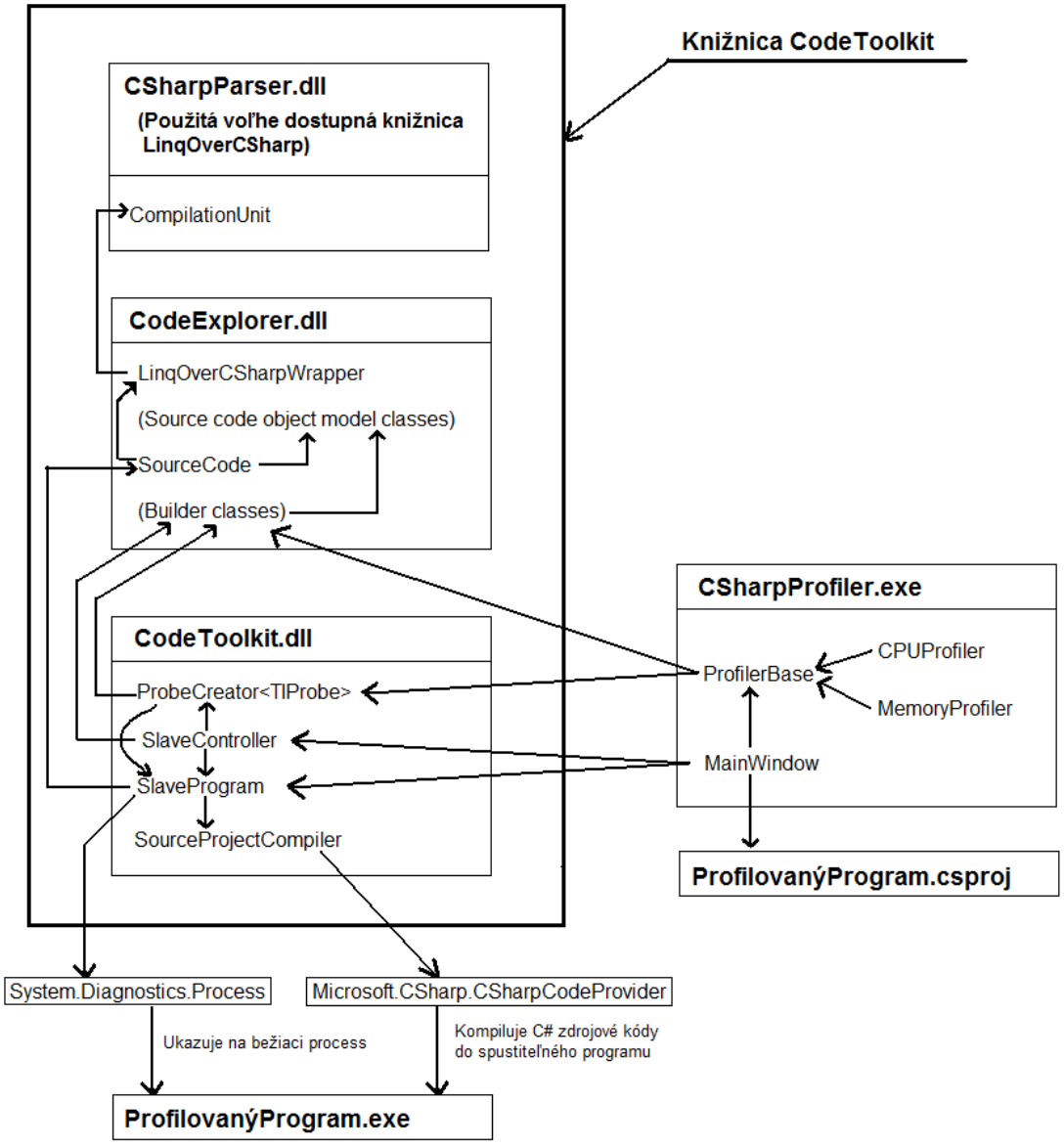
```
string xmlString = "<?xml version='1.0' ?>";
```

Programový kód 4

Toto priradenie však nealokuje žiadnu pamäť, keďže priraďovaný konštantný reťazec bol uložený do pamäte ešte pred spustením programu[2] a toto priradenie len uložilo jeho adresu do objektu typu `System.String`.

5 Implementácia knižnice CodeToolkit

Knižnica Code Toolkit je samostatná univerzálna podporná knižnica určená nie len na tvorbu profilerov založených na inštrumentácii zdrojového kódu, ale i ľubovoľných nástrojov analyzujúci a/alebo modifikujúcich zdrojový kód v jazyku C#. Základné komponenty knižnice Code Toolkit a aplikácie profilera znázorňuje obrázok:



Obr. 24 - Návrh základných komponentov a ich vzťahov

Jednotlivé komponenty budú podrobnejšie rozobrané v nasledujúcich podkapitolách. Pre celkovú predstavu však rovno objasním význam základných častí a ich vzájomnú súvislosť. Knižnica Code Toolkit pozostáva z 3 assemblies[2] – CSharpParser.dll, CodeExpolorer.dll a CodeToolkit.dll. Prvá z vymenovaných je stiahnutá z internetu[29]. Jedná sa o voľne dostupnú knižnicu určenú na parsovanie C# zdrojových súborov. Druhá assembly, CodeExplorer, obsahuje triedy reprezentujúce objektový model programu (viď prílohu A), ale i triedu `LinqOverCSharpWrapper`, ktorej úlohou je parsovanie C# programu do tohto objektového modelu. Ako je na obrázku možné vidieť, toto je jediná trieda, ktorá používa assembly CSharpParser.dll (LinqOverCSharp). Pri parsovaní C# programu si teda parser `LinqOverCSharpWrapper` pomáha využitím tejto existujúcej knižnice. Okrem objektového modelu a parsera obsahuje assembly CodeExplorer i triedu `SourceCode`, ktorá reprezentuje kompletný modelovaný zdrojový program a teda vnútri v nej je možné nájsť jednotlivé prvky modelovaného programu ako napríklad deklarované typy, ich členy, príkazy atď. Trieda `SourceCode` umožňuje zároveň i ukladanie modelu naspäť do zdrojových súborov, ale týmto podrobnostiam sa venuje samostatná časť 5.1.7. Poslednou obzvlášť významnou časťou assembly CodeExplorer je skupina tried typu builder, ktoré verejne poskytujú služby pridávania elementov do modelu programu. Model programu nie je totiž možné verejne priamo modifikovať, ale za týmto účelom je nutné používať špecializované modifikačné triedy.

Tretia assembly CodeToolkit umožňuje prácu s modelovaným programom na vyššej úrovni. Jej jadrom je trieda `SlaveProgram`, ktorá podobne ako `SourceCode` reprezentuje modelovaný program. Presnejšie `SlaveProgram` obsahuje (odkazuje na) `SourceCode` objekt modelovaného programu, ale navyše pridáva funkcionalitu spúšťania, čo zahŕňa uloženie modifikovaného modelu do zdrojových kódov, ich kompiláciu a spustenie vytvoreného skompilovaného programu.

Obr. 24 znázorňuje, že na kompiláciu modifikovaných zdrojových kódov sa používa štandardná trieda `Microsoft.CSharp.CSharpCodeProvier`, kým spustenie (a prípadné ukončenie) je realizované pomocou štandardnej .NET Framework triedy `System.Diagnostics.Process`[4] reprezentujúcej proces operačného systému.

Assembly CodeToolkit tiež obsahuje dve užitočné triedy poskytujúce ďalšiu funkcionalitu na ovládanie modelovaného programu. Generická trieda `ProbeCreator<TIProbe>` umožňuje vkladať do modelovaného programu užívateľskú sondu. Používateľ tejto triedy si môže špecifikovať rozhranie a telo sondy a `ProbeCreator<TIProbe>` sa postará u jej vloženie do modelovaného programu a tiež vytvorí a vráti proxy objekt s rovnakým rozhraním poskytujúci vzdialené volanie metód sondy. Sonda teda umožňuje vzdialene volať užívateľské metódy na bežiacom modelovanom programe. Napokon trieda `SlaveController` poskytuje funkcionalitu pozastavovania bežiaceho modelovaného programu. Z obr. 24 je vidieť, že táto trieda interne používa sondu, aby mohla vzdialene volať metódy pozastavenia a pokračovania. Trieda `SlaveController` ako i predchádzajúce spomenuté triedy obsahujú netriviálnu implementáciu a ich podrobnejší popis je rozobratý v príslušných častiach tejto kapitoly.

5.1 Objektový model C# programu

Assembly CodeExplorer obsahuje typy popisujúce objektový model zdrojového kódu programu v jazyku C#. Pri čítaní nasledujúceho popisu objektového modelu je vhodné súbežne sledovať vyobrazenú hierarchiu tried použitých na reprezentáciu modelu programu v prílohe A. Model programu predstavuje hierarchiu, v ktorej sú najvyššie postavené jednotlivé deklarované typy. V programe jazyka C# je možné deklarovať typ ako `class`, `struct`, `interface`, `enum` alebo `delegate`[12]. Deklarácie týchto typov sú v modely interpretované pomocou objektov príslušnej triedy `Class`, `Struct`,

`Interface`, `Enum`, resp. `Delegate`. Celý zdrojový kód programu potom pozostáva len z kolekcie deklarácii týchto typov a pochopiteľne ich obsahu. V skutočnosti sa takouto interpretáciou môže časť informácie zo zdrojového kódu stratiť. Napríklad atribúty aplikované na celú assembly sa po projekcii do objektového modelu nezachovajú. Súčasná verzia modelu pracuje len s najpoužívanejšou podmnožinou špecifikácie jazyka C# 2.0[12]. Takáto podmnožina jazyka je pre účely profilera dostačujúca, kým zahrnutie kompletnej špecifikácie (obzvlášť v prípade snahy o dosiahnutie špecifikácie C# 3.0) by znamenalo vynaloženie markantného úsilia, ktoré by mohlo byť výhodnejšie prenechať na ďalšie projekty využívajúce túto knižnicu ešte širším spôsobom resp. na samostatný projekt venovaný doplneniu knižnice.

5.1.1 Objektový model typov

Nasledujúci popis odôvodňuje hierarchiu typov reprezentujúcich model programu z prílohy A, preto je minimálne na tomto mieste vhodné sledovať celú situáciu v tejto prílohe. Spoločným znakom štruktúrovaných typov (týmto pojmom sa myslia `class`, `struct` a `interface`) je obsahovanie členov – metódy, konštruktory, položky, vlastnosti, indexery, udalosti a prípadne ďalšie podtypy. Z tohto dôvodu sú triedy, ktoré takéto typy popisujú - `Class`, `Struct` a `Interface`, potomkami triedy `StructuredType` vlastniacej zoznam členov. Triedy `StructuredType`, `Enum` a `Delegate` sú pritom potomkami triedy `TypeDeclaration`, čo je základná abstraktná trieda deklarácie všetkých typov. Trieda `TypeDeclaration` ďalej dedí po triede `PossiblyMember`, čo je abstraktná trieda reprezentujúca element, ktorý by mohol (ale i nemusel) byť členom. Konkrétne potomkami tejto triedy sú všetky triedy reprezentujúce člena (t.j. `Method`, `Constructor`, `Field`, `Property`, `Indexer` a `Event`) a trieda `TypeDeclaration`, ktorá môže ale nemusí byť členom. Znamená to, že slovíčko „Possibly" je použité v názve triedy `PossiblyMember` práve kvôli triede `TypeDeclaration`, ktorá nemusí vždy vyjadrovať člena. Výhodou

takejto hierarchie tried typového modelu je možnosť jednotného spracovania všetkých členov pomocou spoločného predka `PossiblyMember`.

5.1.2 Reprezentácia menného priestoru

Trieda `TypeDeclaration` – predok všetkých tried reprezentujúcich deklaráciu typu obsahuje vlastnosť `ContainingNamespace` ukazujúcu na menný priestor, v ktorom je typ deklarovaný. Avšak táto vlastnosť nemá zmysel v prípade, že typ je členom (podtypom) iného štruktúrovaného typu. V takomto prípade má zmysel vlastnosť `ContainingType` určujúca nadriadený typ.

Menný priestor reprezentujú objekty triedy `Namespace`. Jeden z konštruktorov obsahuje parameter `fullName` špecifikujúci úplný názov menného priestoru. Zdá sa, že tento konštruktora nestojí za zmienku v tomto všeobecnom popise implementácie, keďže potrebné detaily sú v programátorskej dokumentácii. Avšak v skutočnosti tento konštruktor obsahuje menšiu zmienky hodnú optimalizáciu pamäte. Pri neoptimálnom zachádzaní s objektmi typu `Namespace` môže totiž vzniknúť veľké množstvo objektov tohto typu, pričom mnohé z nich zrejme budú obsahovať rovnaký úplný názov menného priestoru. Ak tieto objekty navyše zostanú aktívne (presnejšie dosiahnuteľné z GC koreňov), tak v pamäti zostane „visieť" množstvo inštancií rovnakých reťazcov. CLR (Common Language Runtime – behové prostredie platformy .NET Framework[2]) síce podporuje technológiu „string interning" [2], ktorá sa práve stará o uchovávanie len jedinej kópie unikátneho reťazca, avšak k jej uplatneniu automaticky dochádza len pri konštantných reťazcoch zdrojového kódu. Všetky dynamicky vytvorené reťazce ako napr. výsledok výrazu `"Hello, " + "world!"` nie sú automaticky internované, hoci samotný konštantný reťazec `"world!"` je internovaný s prípadnou ďalšou konštantou `"world!"`.

CLR internovanie dynamicky vytvorených reťazcov sa dá explicitne vynútiť statickou metódou `Intern` typu `String`, avšak CLR internácia má nevýhodu, že internované reťazce zostávajú v pamäti počas celého života aplikácie, teda nikdy nie sú odstránené pomocou GC. Preto spomínaný konštruktor triedy `Namespace` obsahuje optimalizáciu v podobe vlastného zoznamu použitých úplných názvov menných priestorov a pri vytváraní nového názvu sa najprv kontroluje, či sa takýto reťazec už nenachádza v zozname, aby sa mohol použiť len odkaz na reťazec zo zoznamu. Bohužiaľ vzhľadom na to, že použité názvy menných priestorov sú uložené v statickom zozname, tieto názvy tiež zostávajú v pamäti aj keď už neexistujú žiadne inštancie `Namespace`, ktoré by na ne odkazovali[10]. Na rozdiel od CLR internácie tieto reťazce zo zoznamu aspoň zaniknú po odstránení aplikačnej domény[5], čo je šetrnejšie riešenie aspoň v niektorých prípadoch. Avšak je potrebný si uvedomiť, že reťazec, ktorý je parametrom zmieneného konštruktora, sa musel v pamäti tak či tak vytvoriť. Rozdiel je len v tom, že po internácii reťazca (či už tej v CLR alebo implementovanej v `Namespace`) sa odkaz na pôvodný objekt nahradí odkazom zo zoznamu, a teda sa pôvodný objekt uvoľní pre prípadné odstránenie GC kolektorom. Použitá implementácia `Namespace` má teda niektoré nedokonalosti, ale stále si myslím, že je v mnohých prípadoch výhodná.

5.1.3 Používanie typov

Triedy, ktorých predkom je `TypeDeclaration` (t.j. `Class`, `Struct`, `Interface`, `Enum`, `Delegate`), predstavujú deklaráciu príslušného typu. Teda účel ich objektov je popisovať typ (jeho názov, atribúty, modifikátory,

[10] V prípadne potreby by sa tento nedostatok dal obísť buď počítaním referencií alebo ukladaním zoznamu menných priestorov v inštancii `SourceCode`. V druhom prípade by ale príslušná inštancia `SourceCode` musela byť predaná do konštruktora ako parameter, čo by malo nepriaznivý vplyv na jednoduchosť používania menných priestorov.

prípadne hlavičku, zoznam členov atď. podľa príslušného typu). Takéto popisy sú vytvorené len pre typy deklarované priamo v zdrojovom kóde, pre ktorý je objektový model postavený. Všetky ostatné typy pochádzajúce z iných zdrojových kódov (teda z iných assemblies) popis nemajú a je možne sa na ne len odkazovať (v zmysle používať ich). Teda okrem deklarácii (tá by sama o sebe nebola príliš užitočná) sa v programe nachádzajú odkazy na typy. Odkazom na typ sa myslí každé miesto v programe, kde je uvedený názov typu za iným účelom ako je deklarácia príslušného typu. V objektovom modely sú reprezentované objektmi triedy `TypeReference` umožňujúce sa odkazovať na ľubovoľný typ, či už z modelovaného zdrojového kódu alebo z cudzej assembly.

5.1.4 Reprezentácia príkazov a výrazov

Všetky príkazy modelovaného zdrojového kódu sú reprezentované objektmi so základným typom `Statement`. Takto sú modelované všetky príkazy jazyka C# 2.0 okrem príkazov `unsafe`[12] kódu, ktorý zatiaľ nie je podporovaný. Ignorované sú aj príkazy anonymných metód, keďže tieto sú súčasťou výrazov, ktorých rozbor zatiaľ nie je implementovaný.

Osobitne významná je trieda `BlockStatement` reprezentujúca sekvenciu príkazov uzavretých v zloženej zátvorke. Keďže je potomkom abstraktnej triedy `Statement`, môže sa vyskytovať všade, kde patrí príkaz. Navyše aj triedy reprezentujúce členy obsahujúce sekvenciu príkazov (napr. `Method`, `Constructor`, `Property`[11]) uchovávajú túto svoju postupnosť príkazov práve vo vlastnosti typu `BlockStatement`.

---

[11] `Property` sa skladá z dvoch častí `Getter` a `Setter` typu `PropertyAccessor` a až tento typ obsahuje sekvenciu príkazov.

Všetky príkazy obsahujú odkaz na člena, ktorému patria, ale i napríklad zoznam návestí, ktoré sa na danom príkaze nachádzajú. Súčasťou mnohých príkazov je aj jeden, či viac výrazov. Výrazy sú reprezentované triedami s abstraktným predkom `Expression`. Zatiaľ jediným potomkom triedy `Expression` je trieda `UnknownExpression`, ktorý reprezentuje ľubovoľný výraz. Všetky výrazy sú teda reprezentované objektmi tejto triedy. V budúcnosti môžu pribudnúť rôzne špecializované druhy výrazov ako konštanta, lokálna premenná, unárna, binárna, či ternárna operácia a takéto výrazy sa budú môcť do seba skladať a vytvárať zložitejšie výrazy, ale momentálne pre účely profilera nie je takáto podpora výrazov nutná.

5.1.5 Informácia o pôvode modelovaného elementu v zdrojovom kóde

Ako bolo vyššie uvedené deklarácie typov i členy štruktúrovaných typov (metódy, vlastnosti, atď.) majú spoločného predka – triedu `PossiblyMember`. Táto trieda má však ešte priameho predka abstraktnú triedu `CodePart`. Trieda `CodePart` je zároveň predkom príkazov aj výrazov a predstavuje spoločný základ rôznych elementov modelu zdrojového kódu. Obsahuje dve významné vlastnosti `Status` a `Source`. Prvá z menovaných určuje stav pôvodu. Je to výpočtový (`enum`) typ s hodnotami:

`Original` – Element sa zachoval v pôvodnom stave.

`Added` – Element bol pridaný a v pôvodnom zdrojovom kóde sa nenachádzal.

`Changed` – Element sa v pôvodnom kóde nachádzal, ale jeho časť bola zmenená.

Druhá vlastnosť, `Source`, obsahuje informácie o pôvodnom zdrojovom kóde, v ktorom sa nachádzal príslušný element. Táto vlastnosť má zrejme zmysel, len ak nebol element pridaný (`Added`). Je v ňom uchovaný napríklad názov pôvodného súboru, presná pozícia v súbore začiatku a konca elementu

a ďalšie informácie potrebné pre spätné ukladanie modifikovaného modelu do zdrojového kódu ako riadkové odsadenie, či umiestnenie otváracej zloženej zátvorky, čo je potrebné napríklad pre správne vloženie pridaného člena do pôvodnej deklarácie triedy.

5.1.6 Generovanie zdrojového kódu

Triedy reprezentujúce typy, členy, ale i príkazy a výrazy implementujú rozhranie `ISourceCodeGenerator`. Toto rozhranie určuje, že implementujúca trieda obsahuje metódu generujúcu zdrojový kód príslušného elementu. Tejto metóde je možné parametrom predať popis formátovania, ktorý obsahuje niekoľko zaujímavých nastavení podobných možnostiam automatického formátovania v Microsoft Visual Studiu[23]. Napríklad je možné určiť, či má byť otváracia zložená zátvorka umiestnená na novom riadku a to zvlášť u deklarácii typov a zvlášť u príkazových blokov, alebo či sa má vetva `else`, či klauzula `catch` písať na samostatnom riadku. Takisto sa dá zmeniť znak resp. reťazec, ktorý symbolizuje nový riadok.

Pri generovaní zdrojového kódu mnohých elementov sa interne volajú ďalší generátory obsiahnutých elementov. Teda napríklad pri generovaní zdrojového kódu triedy sa volá generovanie jednotlivých členov a členy obsahujúce sekvenciu príkazov zase volajú generovanie `BlockStatement`, ktorý postupne volá generovanie jednotlivých príkazov atď. Vygenerovaný kód jednotlivých elementov sa preto nevracia ako reťazcová hodnota, ale pripája sa na koniec predávaného parametru typu `StringBuilder`. Ak by totiž elementy generovali reťazce, ktoré by sa ďalej spájali do väčších celkov, v pamäti by zostalo veľké množstvo reťazcov, na ktoré by nič neukazovalo. Takéto reťazce by síce GC vyčistil, ale na dočasnú dobu by to spôsobilo väčší tlak na pamäť a zvýšená práca GC by sa mohla prejaviť na výkonnosti generovania obzvlášť pri rozsiahlych zdrojových kódoch.

5.1.7 Trieda SourceCode

Ako už bolo vyššie uvedené na vrchole implementácie objektového modelu je zoznam deklarácií typov, takže takýto zoznam by dostatočne obsiahol celý model zdrojového kódu[12]. Avšak pre pohodlnejšiu prácu existuje trieda `SourceCode`, ktorá zapúzdruje objektový model a pridáva súvisiacu funkcionalitu navyše. Novú inštanciu tejto triedy je možné vytvoriť načítaním (parsovaním) zdrojového programu. Pritom je možné špecifikovať vlastný parser (implementáciou rozhrania `ISourceCodeParser`), alebo nechať použiť základný parser (`LinqOverCSharpWrapper`).

Deklarované typy, ale i ďalšie elementy sa verejne mimo knižnicu nedajú priamo modifikovať, ani neumožňujú pridávanie nových elementov. Interne v rámci knižnice sú však takéto zásahy prípustné. Na uskutočnenie zmeny alebo pridanie elementu je potrebné použiť špecializovanú triedu, ktorej jediný účel je previesť určitý druh úpravy. Pre účely profilera sú zatiaľ k dispozícii napríklad triedy na tvorbu deklarácie triedy (`ClassBuilder`), metódy (`MethodBuilder`), konštruktora (`ConstructorBuilder`) položky (`FieldBuilder`), či ľubovoľného príkazu (`UnknownStatementBuilder`). Objekty týchto tried majú rôzne vlastností určujúce výsledný element. Po ich nastavení je možné zavolať ich vhodnú metódu, ktorá element vloží na požadované miesto v modele.

Veľkou prednosťou triedy `SourceCode` je možnosť uloženia modelu do zdrojového kódu pomocou metódy `Save`. Pri ukladaní sa zachováva pôvodný zdrojový kód, kde je to možné. Ak bol napríklad do originálnej triedy pridaný člen, tak sa trieda uloží na znak presne v pôvodnom tvare a len sa vloží na začiatok do tela triedy zdrojový kód pridaného člena. Tu je vidieť, prečo

[12] Ako už bolo spomenuté súčasná verzia modelu nie je úplná, ale pre účely profilovania dostačuje.

napríklad objekty reprezentujúce štruktúrované typov obsahujú (vo vlastnosti `Source`) aj pozíciu otváracej zloženej zátvorky – pretože pridaný člen sa práve vloží za pozíciu za touto zátvorkou. Podobne sa vkladajú aj príkazy tak, aby okolité príkazy zostali nedotknuté. K tomu je potrebné poznať úplne presné pozície začiatkov a koncov príkazov. Tento problém je spomenutý v ďalšej časti venovanej parsovaniu. Vďaka tomu, že sa pôvodný zdrojový kód zachováva v maximálnej miere, nevadí, že model nie je úplný, pretože časti kódu, ktoré model nepokrýva, aj tak zostanú nedotknuté.

5.1.8 Parsovanie zdrojového kódu

Jedna z najnáročnejších častí tejto práce bolo pripravenie parsera zdrojového kódu. Najprv bolo potrebné zistiť, aká je situácia s voľne dostupnými parsermi jazyka C#. Bohužiaľ, navzdory pôvodným očakávaniam sa mi po dlhodobom hľadaní podarilo nájsť len 2 zmienky hodné voľne dostupné .NET knižnice určené na parsovanie C# programov. Pôvodne bola použitá prvá z knižníc CMicroParser[30]. Po čase sa však ukázalo, že knižnica obsahuje množstvo chýb a čo je ešte horšie, jej vývoj sa zrejme v roku 2006 zastavil, keďže od vtedy nie sú na internetovej stránke jej projektu žiadne aktualizácie. Preto bola táto knižnica nahradená knižnicou LinqOverCSharp[29]. Jedná sa o upravenú verziu parsera vytvoreného generátorom Coco/R[31], v ktorom bolo urobených niekoľko oprav. Parser by si údajne mal poradiť aj so C# 3.0. Bohužiaľ má aj táto knižnica jeden vážny nedostatok kritický pre túto prácu. Získavanie začiatku a konca elementov (hlavne príkazov) je nepresné. Jednotlivé príkazy sa síce sparsujú správne, ale v objektoch reprezentujúcich príkazy je nepresná informácia o pôvodnej pozícii v zdrojovom súbore. Keďže táto informácia je kľúčová pre správne vkladanie pridaných elementov do inak nezmeneného kódu, bolo potrebné urobiť dodatočné parsovanie, ktoré túto informáciu spresní.

Práve toto je úlohou triedy `LinqOverCSharpWrapper`, ktorá vnútorne volá parsovanie knižnice LinqOverCSharp, aby získala objektový model

zdrojového kódu taký, ako ho vytvára samotná knižnica LinqOverCSharp. Takto získaný model následne prekonvertuje na požadovaný model knižnice Code Toolkit. Postupne ako sa pritom generujú objekty príkazov, pôvodný zdrojový kód je opäť parsovaný. Toto parsovanie už ale vie aké príkazy majú nasledovať. Zisťujú sa len ich začiatky a konce, prípadne začiatky a konce ich častí a výrazov. Vzhľadom na rozsiahlosť tejto operácie je trieda `LinqOverCSharpWrapper` rozdelená do 4 súborov postupne zodpovedných za volanie parsera LinqOverCSharp, vytvorenie modelu výrazov, modelu príkazov a modelu typov a ich členov. Trieda `LinqOverCSharpWrapper` obsahuje množstvo konvertujúceho kódu z jedného modelu na druhý a jej časť zodpovedná za budovanie príkazov navyše obsahuje množstvo parsovacieho kódu hľadajúceho pozície jednotlivých častí príkazov. Za zmienku možno stoja dve použité techniky.

Keďže reťazcové konštanty, komentáre a direktívy prekladača značne komplikujú parsovanie, pre uľahčenie sa parsuje radšej maskovaný obsah súboru, kde sú znaky patriace týmto elementom vhodným spôsobom nahradené. Parser tak môže „zabudnúť" na existenciu komentárov, direktív a rôznych typov reťazcových a znakových konštánt. Pomocná metóda, ktorá vytvára takúto masku, musí nájsť nielen znakovú a reťazcovú konštantu v úvodzovkách, ale musí i správne detegovať escape znaky `\"` a `\'` a „verbatim" [12] reťazce jazyka C#, ktoré môžu obsahovať aj znaky nových riadkov.

Druhá použitá technika sa týka priraďovania odkazov na typy. Pri budovaní modelu kódu sa môže často stať, že práve spracovávaný kód používa typ, ktorý je deklarovaný až neskôr. V takom prípade je treba priradenie odkazovaného typu odložiť až kým sa príslušná deklarácia typu nenájde. Avšak trieda `TypeResolver` poskytuje jednoduché riešenie tohto problému. Najprv sa vytvorí objekt typu `TypeResolver`, ktorý si bude

pamätať nevyriešené[13] odkazy na typy a na úplnom konci, keď budú všetky deklarácie známe, vyrieši naraz všetky tieto odkazy. Jednoduchosť spočíva práve v jeho metóde `AddUnresolvedType`, ktorá v parametre dostane nevyriešený odkaz na typ a „akoby zázrakom" hneď vráti objekt reprezentujúci už vyriešeného odkaz. V skutočnosti sa síce vytvorí objekt odkazu a parser ho môže rovno vložiť do modelu, ale objekt zatiaľ neobsahuje správny obsah. `TypeResolver` si odkladá odkazy na všetky takéto objekty, ktoré vytvorí, pričom až na koniec po parsovaní a zavolaní jeho metódy `ResolveTypes` prejde celý ich zoznam a naraz ich vyrieši.

Parser, ktorý vnútorne používa iný parser a jeho vygenerovaný model konvertuje do svojho modelu, pričom doplňuje chýbajúce informácie o pozíciách, nie je úplne ideálny. Toto riešenie je však ešte stále jednoduchšie ako vytvorenie celého C# parsera. V budúcnosti v rámci samostatného projektu bude možno vytvorený i takýto kompletný parser. A práve kvôli flexibilite výmeny je parser pripojený ku Code Toolkit len pomocou rozhrania `ISourceCodeParser`.

5.2 CodeToolkit API

Predchádzajúca podkapitola popisovala implementáciu assembly CodeExplorer, ktorá poskytuje manipuláciu s kódom na nižšej úrovni. Assembly CodeToolkit využíva CodeExplorer a buduje nad ňou ďalšiu funkcionalitu. V assembly CodeExplorer bol objektový model reprezentovaný triedou `SourceCode`, kým CodeToolkit prináša pohľad na modelovaný program pomocou triedy `SlaveProgram`, ktorá dodáva funkcionalitu spúšťania

[13] Pojem „vyriešenie typu" je vytvorený prekladom z anglického „type resolution" a má znamenať prepojenie odkazu na typ s jeho deklaráciou.

modifikovaného kódu. Ďalšie dve triedy, `ProbeCreator` a `SlaveController`, poskytujú komunikáciu s bežiacim modelovaným programom a možnosť jeho pozastavenia. Spolu tieto triedy umožňujú pohodlnú prácu s modelovaným kódom na vyššej úrovni.

5.2.1 Trieda SlaveProgam

Nové inštancie triedy `SlaveProgram` sa vytvárajú načítaním zdrojového programu. Popis tohto zdrojového programu určuje objekt typu `SourceProject`. Tento typ ukrýva v sebe viac ako len zoznam zdrojových súborov. Obsahuje totiž všetky potrebné informácie na skompilovanie a spustenie projektu programu v jazyku C#. Teda jeho súčasťou je i zoznam assemblies, resources a výsledný typ kompilovaného programu (Windows aplikácia, konzolová aplikácia, atď.). Aby bolo možné súbory so zdrojovými kódmi kopírovať do iného adresára s pôvodnou súborovou štruktúrou, tieto súbory majú danú základnú cestu a ich zoznam nepozostáva z absolútnych ciest, ale z relatívnych od tejto základnej cesty. Pre významné zjednodušenie je možné načítať popis zdrojového programu zo zadaného projektového súboru .csproj vývojového prostredia Microsoft Visual Studio. To znamená, že na načítanie C# projektu z Microsoft Visual Studia do objektového modelu (objektu typu `SlaveProgram`) stačia príkazy:

```
SourceProject prj = SourceProject.LoadFromVsProject("...");

SlaveProgram slave = SlaveProgram.Load(prj)
```

Programový kód 5

Na spustenie takéhoto projektu potom stačí zavolať metódu `Run`, ktorá musí najprv projekt skompilovať a následne spustiť vytvorený skompilovaný súbor. Avšak v skutočnosti sa nekompilujú pôvodné zdrojové súbory. Model programu sa môže totiž upravovať. Aby sa zmeny v modele uplatnili, pred kompilovaním sa musí model uložiť do zdrojového kódu, avšak neukladá sa do pôvodných zdrojových súborov (aby sa tieto pôvodné súbory nepoškodili),

ale do kópie v dočasnom adresári. Takže metóda `Run` najprv uloží objektový model programu do pomocných zdrojových súborov, ktoré kopírujú pôvodnú súborovú. Následne sa projekt s týmito novými zdrojovými kódmi skompiluje a nakoniec sa spustí vytvorený súbor. Kompilovanie zabezpečuje trieda `SourceProjectCompiler`, ktorá zaobaľuje volanie kompilátora z triedy `Microsoft.CSharp.CSharpCodeProvider`.

Okrem metódy na spustenie modelovaného programu obsahuje trieda `SlaveProgram` i metódy na jeho ukončenie a tiež umožňuje sledovať, či program stále beží, prípadne inštalovať vlastnú obsluhu udalosti ukončenia programu. Vnútorne je bežiaca aplikácia modelovaného programu reprezentovaná objektom triedy `System.Diagnostics.Process`.

5.2.2 Tvorba sond

Trieda `SlaveProgram` umožňuje spúšťať modifikovaný kód, ale po spustení už s ním neumožňuje žiadnu komunikáciu. Užívateľ knižnice by si túto komunikáciu mohol sám zabezpečiť vložením komunikačného kódu do modelu programu. Avšak keďže táto úloha nie je triviálna a komunikácie bude zrejme dosť častým požiadavkom, preto je súčasťou Code Toolkit i trieda, ktorá dokáže práve túto komunikáciu zabezpečiť.

Trieda `ProbeCreator<TIProbe>` obsahuje metódu (`Inject`) vkladajúcu potrebný kód zabezpečujúci komunikáciu zo strany modelovaného programu. Návratová hodnota tejto metódy implementuje rozhranie špecifikované ako typový argument generickej triedy t.j. `TIProbe`. Na tomto vrátenom objekte sa teda môžu volať metódy z rozhrania `TIProbe`, avšak vykonanie týchto metód sa uskutoční až na strane modifikovaného programu. Presnejšie `TIProbe` definuje rozhranie tzv. sondy. V súčasnej verzii môže sonda obsahovať len metódy. Implementácia týchto metód sa umiestňuje v reťazcovej podobe priamo do rozhrania prostredníctvom atribútu[12] `BodyAttribute`. Takto je možné vkladať do tela i kód,

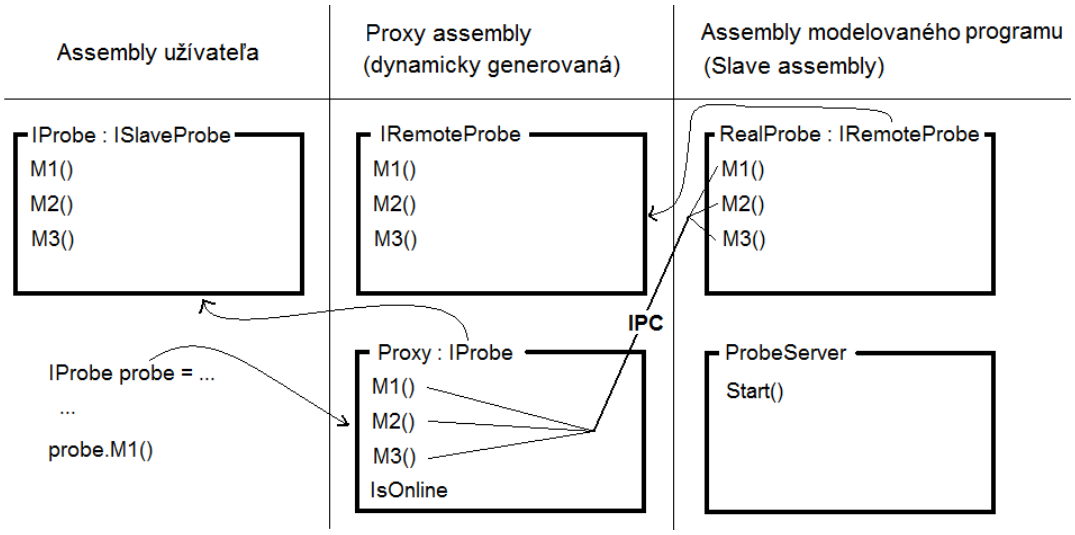
ktorý by nešiel preložiť v programe pracujúcom s Code Toolkit. Metóda `ProbeCreator<TIProbe>.Inject` vloží do modelu programu triedu sondy. Zároveň do neho vloží kód umožňujúci vzdialené volanie metód tejto sondy pomocou IPC (Inter-process communication).

Úplne samostatne sa vygeneruje proxy objekt sondy implementujúci rozhranie `TIProbe`, pričom implementácia jeho metód iba používa IPC klienta na vzdialené volanie príslušnej metódy na skutočnej sonde. Tento proxy objekt sondy je návratovou hodnotou volania metódy `ProbeCreator<TIProbe>.Inject`. Program používajúci Code Toolkit by mohol obsahovať nasledujúce riadky:

```
SlaveProgram slave = ...
IMyProbe myProbe = ProbeCreator<IMyProbe>.Inject(slave);
slave.Run()
myProbe.MyProbingMethod();
```

Programový kód 6

Objekt myProbe je v tejto ukážke proxy objektom sondy. Volanie metódy na poslednom riadku spôsobí vytvorenie IPC klienta a uskutoční vzdialené volanie rovnomennej metódy na skutočnej sonde v modelovanom programe. Celú situáciu znázorňuje obrázok:



Obr. 25 - Vzťah skutočnej sondy a proxy objektu sondy

Užívateľ si musí najprv definovať rozhranie sondy (`IProbe`). Ako už bolo uvedené u každej metódy rozhrania sa musí uviesť i jej telo pomocou atribútu `BodyAttribute`. Metóda `ProbeCreator<IProbe>.Inject` dynamicky vytvorí novú assembly obsahujúcu proxy triedu. Proxy trieda bude implementovať `IProbe` rozhranie sondy, ale jej implementácia bude obsahovať len vzdialené volania. Trieda skutočnej sondy sa vytvorí v modelovanom programe a v ňom sa teda i bude vykonávať skutočná implementácia metód sondy (z atribútu `BodyAttribute`).

Priamočiare riešenie by nechalo skutočnú sondu implementovať rozhranie `IProbe`. Tak by tiež fungovalo, ale bolo by nutné, aby sa modelovanému programu pridala referencia na užívateľskú assembly. Užívateľ tvoriaci svoju assembly by si ale musel dávať pozor, aby nezverejňoval typy, ktoré by mohli spôsobiť konflikt mien s modelovanou assembly. Tomu sa práve Code Toolkit snaží vyhnúť vytvorením rozhrania `IRemoteProbe`, ktoré je komunikačným rozhraním v IPC, takže k modelovanému programu sa musí pridať len referencia na malú generovanú assembly, ktorej všetky typy sú v „anti-kolíznom" mennom priestore (dlhý názov s náhodnými znakmi). Nepríjemným obmedzením bohužiaľ je, že v metódach sondy sa nemôžu prenášať v parametroch a návratovej hodnote iné typy ako „základné" (tým sa myslia primitívne typy a tiež napríklad `System.String`, ale i polia týchto typov), pretože každý zúčastnený typ prenosu musí byť známy na obidvoch stranách, čo by mohlo znamenať nutnosť pridať referenciu na assembly do modelovaného programu a tým sa vystaviť riziku kolízie mien. V závislosti na použitej IPC technológii by navyše užívateľské typy bolo potrebné vhodným spôsobom označiť atribútmi resp. dediť po triede `MarshalByRefObject`[4].

Užívateľom definované rozhranie sondy (`IProbe`) musí vždy dediť po rozhraní `ISlaveProbe`. Toto rozhranie pridáva vlastnosť `IsOnline` indikujúcu pripravenosť IPC komunikácie. Pri implementácii IPC komunikácie sa najviac ponúkali dve možnosti – .NET Remoting[5] a WCF (Windows Communication Foundation[1]). Niekoľko experimentálnych testov však

ukázalo, že .NET Remoting s použitím kanálu IpcChannel[4] je pri malých prenosoch (menej ako 10 KB) často rýchlejší ako WCF s `NetNamedPipeBinding`[4] (WCF trieda reprezentujúca spojenie určené pre IPC). Pri väčších prenosoch sa rýchlejším stáva WCF. Čas na spustenie WCF trvá významne dlhšie – v testoch rádovo stovky milisekúnd, kým pripravenie .NET Remoting servera pre IPC trvá len malé jednotky milisekúnd. Vzhľadom na predpokladané prenosy skôr menšieho charakteru bol nakoniec na sprostredkovanie IPC komunikácie použitý .NET Remoting. V budúcnosti ho bude možné jednoducho nahradiť za WCF, pričom sa rozhranie `IRemoteProbe` bude môcť výhodne použiť ako kontrakt pre toto spojenie.

Jednou zvažovanou možnosťou pôvodne bolo spúšťať modelovaný program ako druhú aplikačnú doménu. Experimentálne som však zistil, že aplikačné domény zdieľajú GC haldu a teda by profilerom namerané údaje o pamäti neboli dostatočne objektívne, keďže by sa v halde nachádzali aj objekty aplikácie profilera.

Triedu `ProbeCreator<TIProbe>` je možné použiť na vytvorenie viacerých sond. Tieto sondy budú existovať nezávisle od seba a každá z nich bude mať vlastný IPC server. Názvy niektorých tried a rozhraní použitých na obrázku (`Proxy`, `IRemoteProbe`, `RealProbe` a `ProbeServer`) sú len ilustratívne. V skutočnosti sa ich mená generujú na základe skutočného názvu rozhrania sondy a pridáva sa poradové číslo sondy.

5.2.3 Pozastavovanie modelovaného programu

Trieda `SlaveController` umožňuje pozastaviť bežiaci modelovaný program. Po zavolaní jej statickej metódy `Inject` s objektom typu `SlaveProgram` v parametri je vrátená inštancia triedy `SlaveController`. Táto metóda `Inject` vloží do modelovaného programu sondu poskytujúcu vzdialené pozastavenie resp. pokračovanie po pozastavení.

Pre správnu činnosť pozastavenia musí ešte zároveň vložiť špeciálny ovládací kus kódu (ďalej bude použitý názov controlling snippet) do kritických miest pôvodného modelu. Objekt `SlaveController`, ktorý vracia metóda `Inject`, obsahuje metódy `Pause` a `Resume`. Obidve volajú rovnomenné metódy na vloženej sonde v modelovanom programe. Metóda sondy `Pause` nastaví špeciálny príznak `PauseRequested` (položka infiltrovanej pomocnej statickej triedy), ale ešte predtým vynuluje druhý príznak `PauseReached` a uzamkne (pomocou metódy `System.Threading.Monitor.Enter`[4]) zámok `PauseWaitingLock` (tiež statická položka infiltrovanej triedy). Príznak `PauseRequested` znamená pre bežiaci program požiadavok na pozastavenie. Na tento príznak pozastavenia reagujú vložené controlling snippets. Ako náhle sa beh programu dostane na controlling snippet, ten zabezpečí pozastavenie a nastaví príznak `PauseReached`, čím signalizuje, že pozastavenie bolo zahájené. Medzitým metóda `Pause` (pochopiteľne vzhľadom na asynchrónnu povahu IPC beží vo vlastnom vlákne) čaká na nastavenie príkazu `PauseReached` a hneď ako zistí, že príznak je nastavený, ukončí sa. Znamená to teda, že volanie metódy `Pause` blokuje beh volajúceho až dovtedy, kým skutočne nenastane pozastavenie, čo je v mnohých situáciách žiadané správanie.

Čakanie na príznak `PauseReached` je implementované pomocou pollingu[35] každých 10 milisekúnd, pretože sa dá predpokladať, že k nastaveniu príznaku dôjde takmer okamžite vzhľadom na hustotu umiestnenia controlling snippets. Metóda `Pause` uzamyká zámok `PauseWaitingLock` a práve na tomto zámku je realizované samotné pozastavenie v controlling snippets, ktoré zjednodušene obsahujú tento kód:

```
if (SlaveController.PauseRequested) {
     SlaveController.PauseReached = true;
     lock(SlaveController.PauseWaitingLock);
}
```

Programový kód 7

Tieto controlling snippets sú vkladané na začiatok každého bloku príkazov a zároveň za každý príkaz s návestím, takže by sa mali vyskytovať vo vnútri každého cyklu. Snáď jediné zdržanie pri čakaní na dosiahnutie controlling snippet by mohlo nastať, ak by bola volaná dlho trvajúca metóda z assembly mimo modelovaný kód. Príkaz `lock`[12] na zámku `PauseWaitingLock` uvedie program do pasívneho čakania, ktoré sa ukončí až vzdialeným volaním metódy `Resume`. Metóda `Resume` najprv vynuluje príznak `IsPauseRequested`, aby sa náhodou hneď po uvoľnení zámku nedostal program do ďalšieho čakania a potom sa uvoľní zámok `PauseWaitingLock` (pomocou `System.Threading.Monitor.Exit`[4]), ktorý ukončí čakanie v controlling snippet a umožní programu pokračovať v normálnom chode.

Takýto spôsob pozastavenia bohužiaľ funguje iba, ak modelovaný program používa len jedno vlákno. V opačnom prípade sa metóda `Pause` vráti už po pozastavení prvého vlákna, ktoré dosiahne controlling snippet. Ostatné vlákna by sa zrejme pozastavili až po čase, prípadne by neboli zastavené vôbec, ak by bola skôr volaná metóda `Resume`. Pre jednoduchosť súčasná verzia Code Toolkit a teda i profiler podporuje len jednovláknové aplikácie. Zdanlivo jednoduché riešenie pozastavenia vo všetkých vláknach by mohlo byť získanie zoznamu všetkých vlákien aplikácie a volanie ich metódy `Suspend` (štandardná .NET trieda `System.Threading.Thread`). Tento spôsob ale môže zanechať objekty v nekonzistnom stave, ak sa napríklad vlákno zastaví uprostred priraďovacej operácie. To môže byť veľmi nežiadúce, keďže po pozastavení môže nasledovať skúmanie stavu objektov pozastaveného programu pomocou sondy.

5.3 Zhrnutie

V tejto kapitole bol predstavený celkový koncept knižnice Code Toolkit a jej jednotlivé časti boli bližšie rozobrané. Ako bolo vidieť, jadrom celej knižnice je objektový model programového kódu v jazyku C#, ktorý je schopný

reprezentovať deklarované typy, ich členy a príkazy jazyka C# 2.0. S pomocou existujúcej knižnice LinqOverCSharp[29] je realizovaný parser zo C# programu do tohto objektového modelu. Model tiež umožňuje spätné ukladanie do zdrojového kódu, pričom sa jednak generuje nový zdrojový kód pridaných elementov, ale zároveň sa zachováva presný pôvodný zdrojový kód, pokiaľ bol v modele nedotknutý.

Celý modelovaný program zapúzdruje trieda `SlaveProgram`. Jej objekty obsahujú prístup k objektovému modelu, ale hlavne umožňujú tento model po modifikácii skompilovať a spustiť ako proces operačného systému. Za behu je s ním potom možné komunikovať vkladaním tzv. sondy, ktorá vytvorí IPC komunikačný kanál pomocou technológie .NET Remoting medzi modelovaným programom a programom používajúcim knižnicu CodeToolkit. Bežiaci program je tiež možné pozastavovať, čo je realizované vkladaním tzv. controlling snippets – kúskov kódu, ktoré na požiadanie blokujú program.

Na manipuláciu s modelom slúžia špecializované refaktorizačné triedy. Pre účely profilera sú zatiaľ implementované tzv. builder triedy na pridávanie rôznych elementov do modelu programu. Builder triedy na vkladanie príkazov sú potom napríklad používané pri vkladaní meracích príkazov profilera, ale presnejší popis tejto aplikácie je uvedený v nasledujúcej kapitole.

6 Implementácia profilera

Užívateľské rozhranie samotnej aplikácie profilera je postavené na WPF (Windows Presentation Foundation). Hoci vytvorenie pohodlného a prehľadného rozhrania vyžaduje nemalé množstvo vývojového úsilia (i keď s použitím WPF je táto práca opäť o niečo jednoduchšia), z hľadiska zameranie tejto práce je však ďaleko podstatnejšie riešenie merania a zhromažďovania profilovaných dát. Meranie časového výkonu a pamäťovej spotreby je realizované pomocou inštrumentácie t.j. pomocou vkladania meracích inštrukcií do zdrojového kódu. Na dosiahnutie inštrumentácie profiler intenzívne využíva vyššie opísanú knižnicu Code Toolkit.

Aplikácia je rozdelená do dvoch režimov – profilovanie spotrebovaného času a profilovanie spotrebovanej pamäte. Obidva režimy pracujú na úrovni príkazu. V režime času to znamená, že užívateľ si zvolí príkazy, ktoré majú byť profilované a pre každý takto zvolený príkaz sa jednotlivo meria čas jeho trvania. V režime pamäte sa u vybraných príkazov meria rozdiel medzi celkovou alokovanou pamäťou pred a po príkaze (zároveň sa meria i počet vykonaných GC kolekcií pre jednotlivé generácie[2]).

Profilovanie času zabezpečuje trieda `CpuProfiler`, kým profilovanie pamäte má na starosti trieda `MemoryProfiler`. Obidve triedy sú potomkami triedy `ProfilerBase` prezentujúcej spoločné rozhranie. Vďaka tomu nie je potrebné rozlišovať, s ktorým profilerom sa práve pracuje, ale stačí pracovať len s rozhraním `ProfilerBase`. V skutočnosti je v triede `ProfilerBase` obsiahnuté okrem spoločného rozhrania i množstvo spoločnej funkcionality. Obidva profilery vkladajú do profilovaného programu meracie príkazy pomocou tried typu builder z assembly CodeExplorer a tiež realizujú príjem nameraných údajov pomocou volania metód vloženej sondy.

Obidve triedy poskytujú 3 typy operácií – vkladanie meracieho a komunikačného kódu do profilovaného programu, komunikácia s bežiacim profilovaným programom a odovzdávanie nameraných výsledkov pre daný príkaz na požiadanie. Aplikácia na pokyn spustenia profilovaného programu nechá najprv objekt triedy vybraného profilera infiltrovať potrebný kód do objektového modelu profilovaného programu a modifikovaný program následne spustí. Namerané údaje sa udržujú v bežiacom profilovanom programe a na žiadosť užívateľa o aktualizáciu sa tieto údaje prenesú do profilera. Po tomto prenose dochádza k postupnému prekresľovaniu príkazov a pri vykresľovaní každého z nich je profiler požiadaný o namerané údaje. Znamená to teda, že najaktuálnejšie namerané údaje sú vždy v bežiacom profilovanom programe. Na žiadosť aktualizácie sa tieto dáta naraz prenesú do profilera, ale jednotlivé hodnoty sa zobrazia až o ne požiada zobrazovacia časť prostredia aplikácie.

6.1 Profiler časovej spotreby

Najzložitejším typom operácie u oboch profilerov je infiltrácia profilovacieho kódu do profilovaného programu. Jednak je potrebné vložiť deklaráciu triedy `StatementData` reprezentujúcu všetky namerané údaje o danom príkaze, ale i deklaráciu statickej triedy `Profiler` obsahujúcu niekoľko položiek, v ktorých sú uložené statické dáta profilera. Profiler času si ku príkazom zaznamenáva počet vykonaní (`HitCount`), celkový akumulovaný čas trvania (`SumTime`) a minimálny resp. maximálny čas trvania (`MinTime` resp. `MaxTime`).

Na meranie času sa používa inštancia štandardnej .NET triedy `System.Diagnostics.Stopwatch`[4], ktorá poskytuje funkciu stopiek. Hodnoty uplynutého času sa ukladajú v jednotkách taktov časovača. Pred každý príkaz, ktorý má byť profilovaný, sa vloží kód na pripočítanie jednotky k počtu vykonaní príslušného príkazu a na spustenie stopiek. Hneď za profilovaným

príkazom sa vloží kód na zastavenie časomiery a získania uplynutého času. Tento čas sa jednak pripočíta k celkovému akumulovanému času príkazu ale dochádza i k prípadnej aktualizácii minima, či maxima. Pochopiteľne operácia spustenia stopiek je posledná vložená operácia pred profilovaným príkazom a prvá operácia po príkaze je ich zastavenie, aby nameraný čas čo najlepšie odpovedal trvaniu tohto príkazu.

V skutočnosti je tento nameraný čas mierne väčší ako čas samotného profilovaného príkazu kvôli zdržaniu pri návrate z metódy spúšťajúcej stopky a pri zavolaní metódy, ktorá ich zastavuje. Na testovanom počítači bol tento rozdielový čas približne $1{,}5\mu s$, čo je prijateľná odchýlka merania. Takto nameraný čas sa anglický označuje ako „wall time" [6] (odvodené od nástenných hodín), pretože ide o čas, ktorý ubehne fyzicky nezávisle od času pridelenému profilovanému vláknu. Takže sa môže stať, že počas vykonávania meraného príkazu operačný systém priradí procesor (resp. jadro procesora) inému vláknu, ale stopky budú stále „bežať na účet" tohto príkazu.

Iný problém by nastal, ak by užívateľ žiadal o profilovanie dvoch príkazov, kde jeden obsahuje druhý (napríklad príkaz cyklu a nejaký príkaz z tela cyklu). V tomto prípade by čas trvania vonkajšieho príkazu bol navýšený o čas trvania merania vnútorného príkazu. Toto navýšenie by mohlo byť veľmi významné a navyše v cykloch by sa tento rozdiel akumuloval. Z tohto dôvodu nie je takýto výber príkazov povolený. Podobne nie je povolené ani časové profilovanie dvoch príkazov z rôznych metód, pretože ak by jeden z príkazov obsahoval volanie metódy, v ktorej sa nachádza ten druhý, tak by bol druhý z nich efektívne obsiahnutý v prvom. Niektoré kombinácie skupín metód (resp. dokonca i párov príkazu a metód) by však zrejme mohli byť povolené v prípade, že neexistuje možnosť zavolania príslušnej metódy z inej metódy resp. z príkazu. To by bolo zaujímavým zadaním problému statickej analýzy kódu, avšak bohužiaľ pre zjednodušenie parsovania súčasná verzia Code Toolkit neposkytuje vhodnú podporu pre analýzu výrazov, ktorá je pre hľadanie dosiahnuteľnosti metód potrebná.

Komunikácia profilera s bežiacim programom je zabezpečená prostredníctvom sondy (viď časť 5.2.2). Teda profiler definuje rozhranie sondy (`ICpuProbe`), ktoré obsahuje metódy `Reset` a `GetData`. Prvá z nich vynuluje údaje o všetkých profilovaných príkazov, kým metóda `GetData` vráti aktuálny stav nameraných údajov. Keďže metódy sondy sú volané asynchrónne zo samostatného vlákna, je potrebné synchronizovať zápis profilovaných dát a prístup sondy k týmto dátam, čo sa uskutočňuje pomocou pozastavenia (viď časť 5.2.3). Pred každým volaním metódy sondy je profilovaný program pozastavený a po jej skončení je naspäť jeho beh obnovený. Pri tomto pozastavení je potrebné zvlášť ošetriť situáciu, v ktorej by pozastavenie nastalo vo vnútri profilovaného príkazu. V takomto prípade sú stopky tiež pozastavované.

V súvislosti s pozastavovaním bolo potrebné sa zamyslieť ešte nad jedným problémom. Do vnútra meraného príkazu sa totiž môže dostať tzv. controlling snippet (viď popis implementácie triedy `SlaveController` v časti 5.2.3) a zvyšovať tak umelo čas jeho trvania. Našťastie Controlling snippet pozostáva z jednoduchej podmienky `if`, ktorej kladná vetva sa vykoná len v prípade požiadavku na pozastavenie (v takom prípade sa stopky tiež pozastavujú). K zdržaniu teda dochádza len pri vyhodnocovaní booleanovskej statickej položky v príkaze `if`, čo je celkom zanedbateľný čas. Dokonca i keď k týmto testom dochádza mnohokrát v cykle, tento rozdiel sa typicky akumuluje podstatne pomalšie ako pôvodné príkazy cyklu. Keďže viaceré vlákna bežiaceho programu (okrem spomínaného vlákna sondy) nie sú na prístup k profilovacím údajom synchronizované, profilovanie je funkčné len s jednovláknovými programami.

6.2 Profiler pamäťovej spotreby

Princíp fungovania pamäťového profilera je rovnaký ako u profilera času vrátane vloženej triedy `StatementData` reprezentujúcej merané údaje jedného príkazu, vloženej statickej triedy `Profiler` nosiacej stav profilovania,

komunikácie prostredníctvom sondy s metódami `Reset` a `GetData`, či pozastavovania programu kvôli synchronizácii prístupu k dátam profilovaných príkazov. Pri meraní pamäte odpadajú všetky problémy týkajúce sa nežiaducich zdržaní vo vnútri meraného príkazu. Vďaka tomu je povolené merať aj príkazy z rôznych metód naraz a i príkazy, ktoré jeden druhého obsahujú.

Pamäťový profiler pri príkaze meria rozdiel alokovanej pamäte haldy pred a po jeho vykonaní (údaj získaný statickou metódou `GetTotalMemory` štandardnej .NET triedy `System.GC`[4]) a počet kolekcií GC počas vykonávania príkazu (statická metóda `CollectionCount` tiež triedy `System.GC`). Podobne ako pri profilovaní času trvania príkazu aj pri meraní spotrebovanej pamäti sa opäť počíta celková akumulovaná hodnota, minimum a tiež maximum nameraných hodnôt jednotlivých vykonaní príkazu.

Pôvodne bola plánovaná aj implementácia sledovania GC koreňov ako je opísané v podkapitole 2.2. Profiler by potom mohol za behu poskytovať informácie o jednotlivých žijúcich objektoch, graf ich vzájomného odkazovania, ich veľkosti a dokonca by mohli byť v zdrojovom kóde vyznačené položky a lokálne premenné, z ktorých je možné dosiahnuť vybrané objekty. Táto funkcionalita sledovania GC koreňov by však predstavovala ďalšiu komplikovane implementovanú (hoci snáď veľmi jednoducho používateľnú) komponentu knižnice Code Toolkit. Kvôli časovým možnostiam, ktoré boli vyčerpané už aj tak komplexnou implementáciou základov knižnice, bolo nakoniec od sledovania GC koreňov upustené. Táto určite zaujímavá funkcionalita by sa však mohla objaviť v budúcej verzii.

7 Záver

Pôvodným cieľom bakalárskej práce bolo vytvorenie profilera programov napísaných v jazyku C#. Tento cieľ bol dosiahnutý. Vytvorený profiler umožňuje merať čas trvania vybraných príkazov a tiež ich spotrebovanú pamäť. Demonštratívne príklady použitia v podkapitole 4.2 a pozorovania získané meraniami v týchto príkladoch ukazujú, že profiler môže ponúknuť zaujímavé a hodnotné informácie o skúmanom programe. Na druhej strane pochopiteľne profiler vytvorený v tejto práci neobsahuje takú bohatú funkcionalitu, akú je možné nájsť v niektorých komplexných komerčných programoch, ktoré sú vyvíjané v tímoch a zrejme i boli vyvíjané podstatne dlhší čas. Medzi nedostatky vytvoreného C# profilera patrí napríklad obmedzenie len na jednovláknové aplikácie, neschopnosť profilovania projektov obsahujúcich resources, či možnosť voliť meranie času trvania príkazov súčasne len v jednej metóde a to len tak, aby žiadny vybraný príkaz nebol vnorený v inom vybranom príkaze. Avšak napriek týmto obmedzeniam vzniknutý profiler stále poskytuje i funkcionalitu, s ktorou som sa nestretol u žiadneho voľne dostupného profilera. Vďaka svojej schopnosti merať spotrebovanú pamäť jednotlivých príkazov môže dokonca dopĺňať mnohé komerčné profilery.

I keď by sa našlo uplatnenie pre samotný profiler, najväčší význam tejto práce však prináša knižnica Code Toolkit, ktorá pôvodne vznikala ako pomocná knižnica profilera poskytujúca manipuláciu so zdrojovým kódom. Počas vývoja tejto práce bolo potrebné vyriešiť reprezentáciu zdrojového kódu v objektovom modele, parsovanie zdrojových súborov do tohto modelu a ukladanie tohto modelu po zmodifikovaní naspäť do zdrojových súborov. Táto funkcionalita bola umiestnená do knižnice Code Toolkit tak, aby mohla byť opakovane jednoduchým spôsobom využívaná v ďalších nástrojov analyzujúcich a/alebo modifikujúcich zdrojový kód. Knižnica navyše poskytuje vysoko úrovňové nástroje poskytujúce načítanie C# projektu z programu Microsoft Visual Studio do objektového modelu pomocou dvoch jednoduchých volaní metód, či skompilovanie a spustenie modifikovaného modelu pomocou volania jedinej

metódy. Medzi jej ďalšie služby patrí pohodlné zriadenie IPC komunikácie s bežiacim modelovaným programom vďaka možnosti vytvorenia sondy s užívateľskými metódami, ktorá sa automaticky infiltruje do objektového modelu programu a zároveň sa na klientskej strane vygeneruje proxy objekt umožňujúci vzdialené volanie metód tejto sondy.

S touto aparatúrou je možné Code Toolkit výhodne využiť v ďalších aplikáciách pracujúcich so zdrojovým kódom. Avšak i táto knižnica má stále množstvo miest poskytujúcich priestor na vylepšenie. Objektový model programu nepokrýva všetky elementy jazyka C#, pričom najviac chýba modelová reprezentácia výrazov. Model je možné meniť len pridávaním ďalších elementov a i obmedzenie profilera na jednovláknové aplikácie a podpora jazyka C# maximálne verzie 2.0 pramení z obmedzenia v knižnici Code Toolkit. V každom prípade bol však v tejto práci vytvorený plnohodnotne použiteľný základ knižnice umožňujúcej pohodlnú manipuláciu so zdrojovým kódom. V budúcnosti bude možné Code Toolkit doplniť v týchto neimplementovaných miestach a tým ešte viac rozšíriť možnosti jej použitia.

Referencie

[1] Web komunity Windows Communication Foundation, Microsoft Corp., 2007
http://netfx3.com/content/WCFHome.aspx

[2] Richter, J.: CLR via C# (Second edition), Microsoft press, 2006

[3] Wikipedia: Inter-process communication
http://en.wikipedia.org/wiki/Inter-process_communication

[4] Microsoft Developer Network, Microsoft Corporation, 2008
http://msdn.microsoft.com/

[5] Nagel Ch.: Professional C# 2005, Wrox Press, 2005.

[6] Wikipedia: Wall clock time
http://en.wikipedia.org/wiki/Wall_time

[7] Domáca stránka projektu Mono
http://www.mono-project.com/

[8] Domáca stránka knižnice Cecil
http://www.mono-project.com/Cecil

[9] Wikipedia: Performance analysis
http://en.wikipedia.org/wiki/Profiler_(computer_science)

[10] Wikipedia: Code coverage
http://en.wikipedia.org/wiki/Code_coverage

[11] Wikipedia: Lock (computer science)
http://en.wikipedia.org/wiki/Lock_(computer_science)

[12] Špecifikácia jazyka C#, štandard ECMA-334, Ecma International, 2006
http://www.ecma-international.org/publications/standards/Ecma-334.htm

[13] Internetový vyhľadávač, Google Inc., 2008
http://www.google.com/

[14] Prof-It for C#, prof-it@gmx.net
http://dotnet.jku.at/projects/Prof-It/

[15] NProfiler, Jon Shute, 2002
http://sourceforge.net/projects/nprofiler/

[16] ProfileSharp 1.3, SoftProdigy, 2008
http://www.softprodigy.com/

[17] Web na sťahovanie softwaru spoločnosti Microsoft, Microsoft Corp., 2008
http://www.microsoft.com/downloads/

[18] .NET Profiler, YourKit, LLC, 2008
http://www.yourkit.com/dotnet/

[19] .NET Memory Profiler, SciTech Software AB, 2008
http://memprofiler.com/

[20] dotTrace 3.1, JetBrains, 2008
http://www.jetbrains.com/profiler/

[21] AQtime, AutomatedQA, Corp., 2008
http://www.automatedqa.com/products/aqtime/

[22] ANTS Profiler 3.2, Red Gate® Software Ltd, 2008
http://www.red-gate.com/products/ants_profiler/

[23] Visual Studio 2008, Mircrosoft Corporation, 2008
http://msdn.microsoft.com/en-us/vstudio/

[24] Licencia GNU General Public License, Free Software Foundation, Inc., 2007
http://www.gnu.org/licenses/gpl.html

[25] Wikipedia: Prime number
http://en.wikipedia.org/wiki/Prime_number

[26] Wikipeda: Sieve of Eratosthenes
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

[27] Design Guidelines for Developing Class Libraries, Microsoft Corp., 2008
http://msdn.microsoft.com/en-us/library/ms229042.aspx

[28] Web komunity Windows Presentation Foundation, Microsoft Corp., 2008
http://windowsclient.net/default.aspx

[29] LinqOverCSharp – knižnica C# parsera, 2008
http://www.codeplex.com/LinqOverCSharp

[30] C# Parser, Omer van Kloeten, 2007
http://www.codeplex.com/csparser

[31] Compiler Generator Coco/R
Institut für Systemsoftware, Johannes Kepler Universität Linz
http://www.ssw.uni-linz.ac.at/coco/

[32] Wikipedia: Component Object Model
http://en.wikipedia.org/wiki/Component_Object_Model

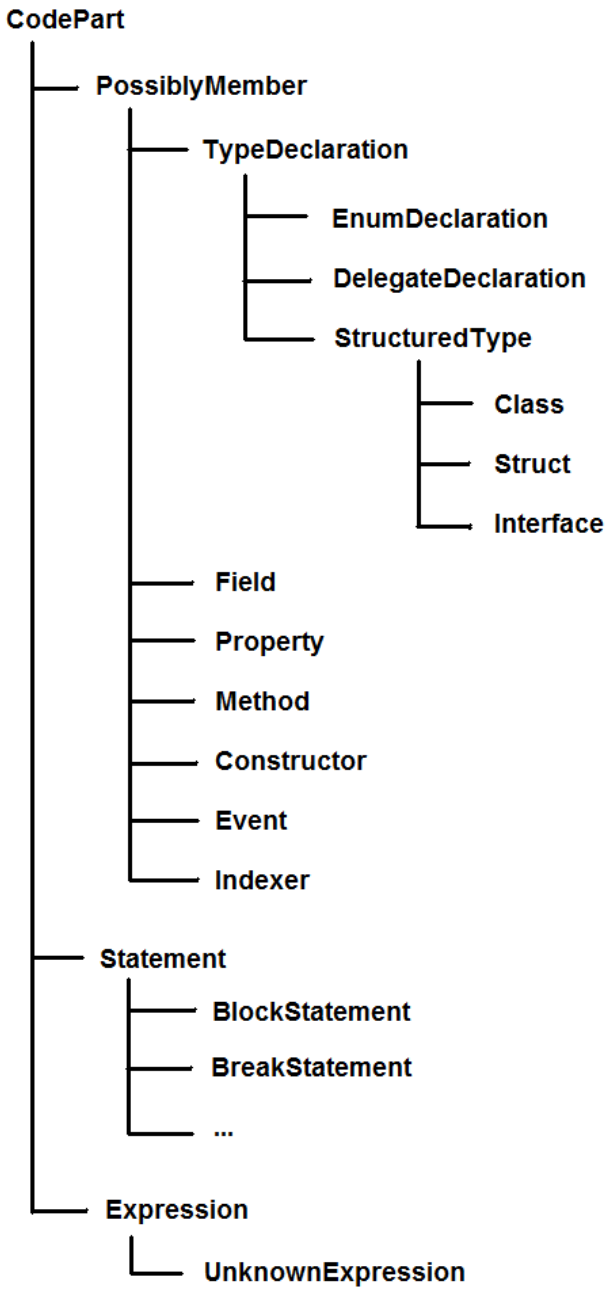
[33] Wikipedia: XML
http://en.wikipedia.org/wiki/Xml

[34] Wikipedia: Immutable object
http://en.wikipedia.org/wiki/Immutable_object

[35] Wikipedia: Polling (computer science)
http://en.wikipedia.org/wiki/Polling_(computer_science)

Príloha A: Hierarchia tried použitých na reprezentáciu zdrojového kódu

Program v jazyku C# je v knižnici CodeToolkit reprezentovaný objektovým modelom, ktorý je bližšie opísaný v podkapitole 5.1. Táto príloha zobrazuje hierarchiu dedičnosti tried, ktoré sa v tomto objektovom modele používajú na reprezentáciu hlavných elementov programu:



Príloha B: Obsah priloženého CD-ROM

Súčasťou tejto bakalárskej práce je i priložené CD-ROM s nasledujúcim obsahom:

/.NET Framework 3.5 Setup/dotnetfx35.exe

- Inštalátor .NET Framework 3.5.

/Profiler/

- Adresár so spustiteľným programom C# Profilera.

/ProfilingSample/

- Zdrojové kódy ukážkového programu na profilovanie.

/Sources/

- Zdrojové kódy knižnice Code Toolkit a programu C# Profiler.

/Bakalarska praca.pdf

- Text tejto práce.

/Documentation.chm

- Vygenerovaná dokumentácia zdrojových kódov.

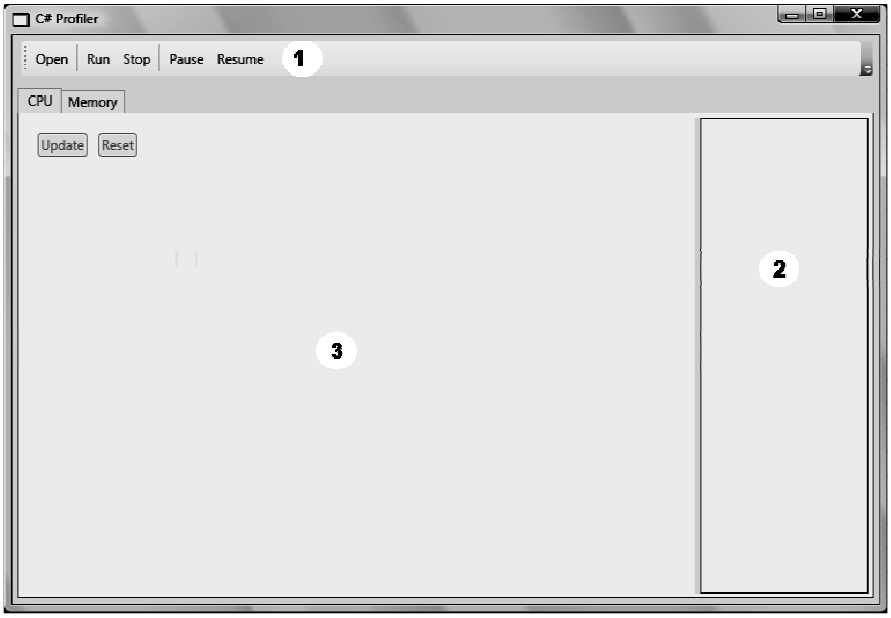
Príloha C: Popis inštalácie

Aplikácia C# Profiler pracuje na platforme .NET Framework 3.5 a z toho vyplývajú požiadavky, ktoré sú minimálne Windows XP so Service Pack 2 a samozrejme nainštalovaný .NET Framework 3.5. Inštalátor tohto .NET Frameworku 3.5 sa nachádza na priloženom CD-ROM v adresári „*.NET Framework 3.5 Setup*".

K programu nie je dodávaný inštalátor, keďže inštalácia nie je potrebná. Pre spustenie stačí skopírovať súbory `CSharpProfiler.exe`, `CodeToolkit.dll`, `CodeExplorer.dll` a `CSharpParser.dll` (nachádzajú sa v adresári „*Profiler*" na priloženom CD-ROM) do jedného adresára a v ňom spustiť prvý z vymenovaných súborov. V tomto adresári musí mať program právo na vytváranie podadresárov a súborov, pretože po otvorení profilovaného programu sa tu vytvorí pomocný adresár, do ktorého sa ukladajú napríklad modifikované zdrojové kódy.

Príloha D: Užívateľský manuál

Aplikácia profilera pozostáva z jediného hlavného okna, ktoré sa skladá z 3 oblastí ako ukazuje obrázok:



Obr. 26 – Hlavné okno profilera

D.1 Základné ovládanie

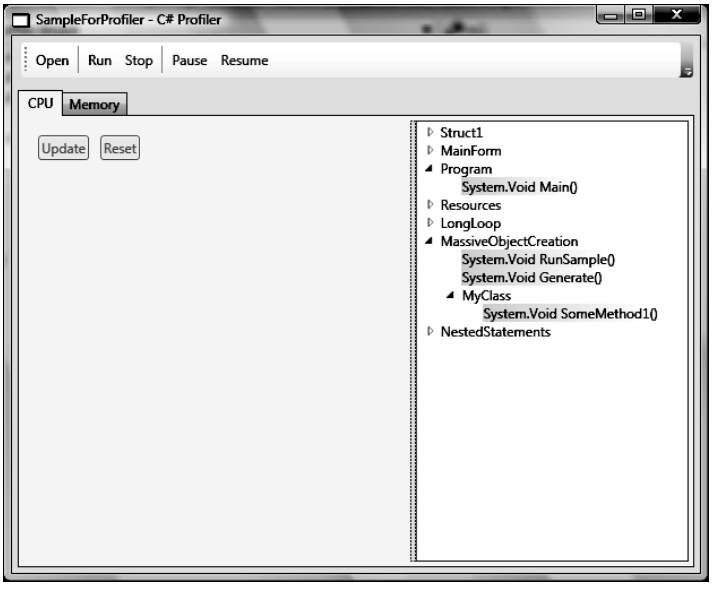
Oblasť označená číslom 1 na obr. 26 obsahuje základné ovládacie tlačítka umožňujúce postupne otvorenie projektu, spustenie a ukončenie profilovanej aplikácie a napokon pozastavenie resp. pokračovanie po pozastavení tak ako naznačujú jednotlivé nápisy.

Na začiatku nie je otvorený žiadny projekt. Otvárať je možné C# projekty programu Microsoft Visual Studio 2003, 2005 a 2008, teda súbory s príponou .csproj. Nie je možné otvárať projekty, ktoré odkazujú na iné projekty. Práca s projektmi, ktoré obsahujú resources[4], môže byť problematická, keďže v súčasnej verzii nie sú resources podporované. Projekt bude síce úspešne otvorený, ale pri spustení sa k nemu resources nekompilujú, teda zrejme budú profilovanej aplikácii počas jej behu chýbať. Názov otvoreného projektu sa tiež objaví v titulku okna.

Spúšťacie tlačítko spustí profilovanú aplikáciu. V bežiacej aplikácii sa automaticky uskutočňujú požadované merania. Tlačítko „Stop" profilovanú aplikáciu okamžite „násilne" ukončí. Naopak operácia „Pause" je k aplikácii veľmi šetrná. K pozastaveniu dôjde, až keď sa program dostane do bezpečného bodu. To môže chvíľu trvať. Počas potenciálne dlhšie trvajúcich operácii ako je táto sa kvôli bezpečnosti dočasne zneprístupnia všetky ovládacie prvky hlavného okna a v strede sa zobrazí nápis upozorňujúci na vykonávanie operácie.

D.2 Pohľad na metódy a konštruktory

Hierarchia tried a štruktúr (`struct`) deklarovaných v otvorenom projekte sa zobrazuje v oblasti označenej číslom 2 na obr. 26. Jedná sa o klasický stromový pohľad. Pod zložkou reprezentujúcou triedu (resp. štruktúru) sa nachádzajú deklarované metódy prípadne konštruktory. Avšak zobrazujú sa len tie triedy (štruktúry), ktoré majú explicitne deklarovanú aspoň jednu metódu alebo konštruktor. Príklad stromovej štruktúry ukazuje obr. 27:



Obr. 27 – Stromová štruktúra typov

Dvojitým kliknutím na metódu resp. konštruktor sa zobrazí jeho zdrojový súbor.

D.3 Výber profilera a príkazov na meranie

V tretej oblasti na obr. 26 sa zobrazuje otvorený zdrojový súbor. Zároveň je ale možné si všimnúť, že táto oblasť spolu so stromovou štruktúrou typov je umiestnená v prepínacom panely. Panel umožňuje voliť medzi „CPU" a „Memory" t.j. profilovaním času alebo pamäte. Naraz je možné pracovať iba v jednom z týchto dvoch režimoch.

Niektoré príkazy v otvorenom zdrojovom kóde majú pri sebe zaškrtávacie políčko. Zaškrtnutie tohto políčka určí, či sa má uskutočniť meranie tohto príkazu pri ďalšom spustení profilovaného programu. Pri profilovaní pamäte majú takéto políčko všetky príkazy metód a konštruktorov mimo príkazov, ktoré sú súčasťou anonymných metód a príkazov v `unsafe` bloku. Pri profilovaní času po zaškrtnutí políčka navyše zmiznú políčka príkazov z iných metód a konštruktorov a tiež všetkých príkazov, ktoré sú vnorené alebo nadriadené. Teda naraz je možné merať časy príkazov len v jednej metóde (resp. v konštruktore) a pritom žiadne dva merané príkazy nesmú byť jeden v druhom vnorený, aby sa zabránilo skresleniu merania trvania nadriadeného príkazu kvôli meracej réžii vnoreného príkazu. Pridržaním klávesy **Ctrl** pri zaškrtávaní a odškrtávaní sa označia resp. odznačia všetky príkazy v bloku vrátané vnorených. Samozrejme v prípade profilera časov dochádza len k povoleným označeniam.

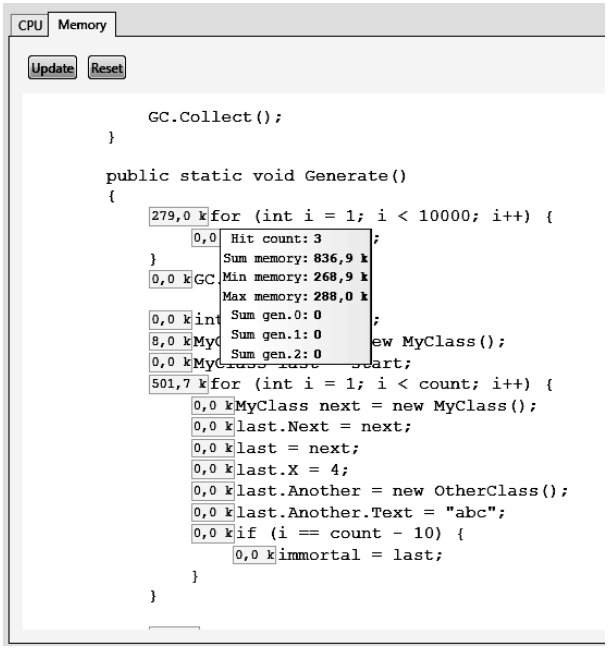
Pri meraní pamäte je možné označovať príkazy naraz z viacerých zdrojových súborov. Pre lepší prehľad sa v stromovej štruktúre tried tučne zvýrazňujú triedy, štruktúry, metódy a konštruktory, ktoré obsahujú aspoň jeden označený príkaz.

D.4 Ovládanie profilera a zobrazovanie výsledkov

Po označení požadovaných príkazov a spustení programu pôvodné zaškrtávacie tlačítka zmiznú a označené príkazy majú na ich mieste zobrazený výsledok merania. V prípade, že daný príkaz ešte nebol vykonaný ani raz,

zobrazí sa len text „(no hit)“. V opačnom prípade sa zobrazí priemerný čas trvania (v milisekundách) resp. priemerná pamäťová spotreba (v kilobajtoch) jednotlivých vykonaní daného príkazu. Namerané údaje sa však neaktualizujú automaticky. K aktualizácii dochádza až po stlačení tlačítka „Update“. Tlačítko „Reset“ umožňuje vynulovať všetky namerané údaje. Obidve tieto tlačítka pre svoje korektné fungovania pozastavujú profilovaný program. K tomu sa používa rovnaká funkcionalita ako vo vyššie popisovanom pozastavovaní pomocou tlačítka „Pause“. Pri aktualizácii i vynulovaní nameraných údajov sa teda tiež musí čakať, kým program dosiahne bezpečný bod pre pozastavenie.

Ak bol meraný príkaz aspoň raz vykonaný a teda pri sebe obsahuje svoj priemerný čas trvania resp. priemernú spotrebovanú pamäť, po premiestnení kurzora myši nad túto zobrazenú priemernú hodnotu sa zobrazí malé okno s ďalšími nameranými údajmi daného príkazu. V prípade profilera času sa zobrazí počet vykonaní príkazu a celkový, minimálny a maximálny čas jeho trvania. V prípade profilera pamäte sa okrem počtu vykonaní príkazu zobrazí celková, minimálna a maximálna spotrebovaná pamäť jeho jednotlivých vykonaní a tiež celkový počet kolekcií GC pre jednotlivé generácie, ktoré sa uskutočnili počas vykonávania daného príkazu:



Obr. 28 – Detail nameraných údajov